UML ASSISTED VISUAL DEBUGGING FOR DISTRIBUTED SYSTEMS

THESIS

Benjamin R. Musial, Flight Lieutenant, RAAF

AFIT/GCS/ENG/03-12

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/03-12

UML ASSISTED VISUAL DEBUGGING FOR DISTRIBUTED SYSTEMS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Benjamin R. Musial, B. E. (Hons.)

Flight Lieutenant, RAAF

March 2003

AFIT/GCS/ENG/03-12

UML ASSISTED VISUAL DEBUGGING FOR DISTRIBUTED SYSTEMS

Benjamin R. Musial, B. E. (Hons.)
Flight Lieutenant, RAAF

Approved:

[illegible] 7 MAR 03

Timothy M. Jacobs, LtCol, USAF — Date
Chairman

[illegible] 7 MAR 03

Karl S. Mathias, LtCol, USAF — Date
Committee member

[illegible] 7 MAR 03

Gary B. Lamont, PhD — Date
Committee member

Acknowledgements

More than anyone else, I would like to thank my wife for keeping me sane and helping me to see “the light at the end of the tunnel” throughout the AFIT experience. Her invaluable visits and our vacations from schoolwork kept me motivated and focused while she organized our wedding. I would also like to thank the rest of my family for their constant support.

I would like to thank the AFIT faculty for sharing their knowledge and experience with me. They have made the last 18 months of work worthwhile. Special recognition goes to Lt Col Timothy Jacobs for his guidance and support in our publications.

I would also like to thank my fellow classmates. Their assistance and comradeship during the course was invaluable.

Benjamin R. Musial

Table Of Contents

Page

Table Of Figures

Page

AFIT/GCS/ENG/03-12

Abstract

The DOD is developing a Joint Battlespace Infosphere, linking a large number of data sources and user applications. To assist in this process, debugging and analysis tools are required. Software debugging is an extremely difficult cognitive process requiring comprehension of the overall application behavior, along with detailed understanding of specific application components. This is further complicated with distributed systems by the addition of other programs, their large size and synchronization issues. Typical debuggers provide inadequate support for this process, focusing primarily on the details accessible through source code. To overcome this deficiency, this research links the dynamic program execution state to a Unified Modeling Language (UML) class diagram that is reverse-engineered from data accessed within the Java Platform Debug Architecture. This research uses focus + context, graph layout, and color encoding techniques to enhance the standard UML diagram. These techniques organize and present objects and events in a manner that facilitates analysis of system behavior. High-level abstractions commonly used in system design support debugging while maintaining access to low-level details with an interactive display. The user is also able to monitor the control flow through highlighting of the relevant object and method in the display.

UML ASSISTED VISUAL DEBUGGING FOR DISTRIBUTED SYSTEMS

1. Research Introduction

1.1 Introduction

The DOD maintains a large number of databases and other information sources that are both geographically displaced and incompatible in the communication protocols they employ. The Joint Battlespace Infosphere (JBI) proposes a "combat information management system that provides individual users with the specific information required for their functional responsibilities during crisis or conflict [USA00]." The JBI provides a system to integrate a wide variety of data sources and distribute information that is both relevant and at the right level of detail to all DOD users. The data will be available from many servers on a network known as the Global Information Grid. This network is to connect all services required by the JBI to access the information and deliver it back to the user. In essence, the JBI will be a large distributed database system.

Consideration of the description of the JBI leads to some idea of the complexity of this system. The DOD has hundreds of data sources using many different protocols. Combining the number of types of data sources and protocols with the number of possible outputs from the system leads to literally millions of possible combinations of data transformations to deal with. Connecting multiple data sources in a distributed environment is difficult at the best of times. The complexity of the JBI as described suggests that debugging and analysis tools would be of great assistance during the integration of data sources and sinks.

1.2 Background

A variety of debugging systems are available to deal with distributed systems, however few handle the variety of operating systems required within the JBI. Prior work by AFIT in this field established a Java based system primarily concerned with the analysis of distributed systems based on inter-agent communication [KIL02]. The features of Java ensure easy platform independence in distributed monitoring systems.

Debugging distributed systems produces large amounts of data, lending itself to improvement through visualization. Visualization increases the human-computer data bandwidth compared to traditional character reading. In addition, it allows for more efficient pre-cognitive processing of the data requiring less effort from the user. As such, many effective distributed debugging and analysis tools rely on visualization techniques [BOW94, JER97].

One of the main views presented in distributed debugging tools is similar to a sequence diagram [JER97, APP93, AMA99]. Sequence diagrams show control flow in programs. Recently developed applications using this model allow grouping of sub-components into higher-level systems in the architecture [JER97].

Virtually all tools in this domain focus on the communication between modules described by size, delivery time and location (sender and receiver). Telles states that one of the most important aspects of debugging is the ability to consider the big picture [TEL01]. The user should base their hypothesis of the bug's cause and location on the overall system behavior. For this reason, the inclusion of UML modeling to represent the system is considered valuable. As discussed, the sequence diagram provides assistance in debugging in many existing tools but is unable to provide an overview of the total system. Class diagrams are able to describe the system adequately in most cases. Software engineers frequently use class diagrams during system design and these are the most commonly used UML diagram [COO99]. This thesis investigation intends to show that the use of dynamic UML object diagrams would assist the user during the debugging process.

1.3 Research Focus

This thesis effort focuses on developing more effective methods for applying visualization to distributed debugging. The researcher implements these methods in a Java based system to demonstrate the capabilities and assess their effectiveness. Much of this work centers on methodologies that incorporate existing technologies in new ways, such as use of UML in debugging. As with other tools, it is necessary to view the system from various levels of abstraction, allowing the user to scrutinize and correct likely error causing modules. The use of multiple levels of abstraction allows the user to gain a global view of the system while having access to detailed information essential to effective debugging.

The domain of the solution is limited to Java based applications to simplify data extraction in the debugging process. Java libraries including the Java Platform Debug Architecture (JPDA) are utilized for the automatic run-time extraction of data. The Java2D libraries combine with additional code to implement the visualization component of the system.

1.3.1 Objectives

The primary goal of this thesis investigation is to develop methodologies to allow more effective debugging of distributed systems. This research must meet several requirements in order to achieve the primary goal. Initially, the debugging process is analyzed to determine key techniques and requirements. These requirements and techniques are adapted where necessary to ensure they are relevant to debugging of distributed systems. The main requirement in the debugging process is to gain an understanding of system behavior including both static and dynamic information [TEL01]. Wide varieties of views are required in order to present the user with large amounts of data as required.

Any system with this goal should automate the method of data extraction, as large amounts of data are required for this debugging process. Displaying the program structure aids in the presentation of data. The user should be able to select the monitoring level desired for a data type. For example, user interest does not require access to all the information for many classes at any particular time; the display of the system should reflect this. In addition, the system should display execution path of the program for the user to see in the abstract views.

It is essential that the person performing the debugging have access to various levels of abstraction to facilitate program understanding. This user should have views ranging from source code to architectural views showing where the current code segment fits in with the rest of the system and the relationships it maintains with those modules surrounding it.

A display of the program structure is an important part of the visualization created for the debugging process. It is envisaged that the ability to see where debug data is coming from in a well-known layout representation assists the user in detecting program errors. It is essential that the layout be in a well-known form to take advantage of inherent user knowledge of layout, for example showing inheritance relationships vertically. The program layout information should be rapidly reverse-engineered without the need for source code as it may not be available or current. Automation simplifies use of the system for the user.

As network performance can influence the operation of a distributed system, it is desirable to have important network parameters alongside debug and program structure information. With this approach, the user can match quickly and easily locations having a high probability of causing an error with those encountering unusual network conditions. For example, should a link or server go down and cause errors in the system, the user should easily be able to determine a probable error location through some visual means.

Automatic detection of possible program flaws would be a highly useful feature in the system. It is desirable that one of these techniques be included in the debugging system to simplify detection of bugs in the system under observation.

As with all visualization systems, the system must address several key issues such as the use of color, patterns and movement. Effective utilization of these visual identifiers will be required to maximize the benefits of the system. Other visualization techniques are incorporated to allow a global view with detail for areas of interest. These techniques enhance visual processing by the user.

This system must satisfy a variety of smaller requirements in order to achieve the primary objective of this research. The most important aspects of debugging for distributed systems must be determined and incorporated in the system with automatic Graphical User Interface (GUI) driven access to the data. The system should combine time-domain information with network parameters, program structure and debug data to provide the best view of system operation. Automatic detection should highlight those areas that are most likely the cause of program error and visualization methods should take advantage of pre-cognitive processing where possible.

1.3.2 Limitations

A wide variety of data sources and applications both in and outside of the DOD are currently Java based with the trend likely to continue. For ease of implementation, this research limits its scope to those systems that are Java-based. Without this limitation, many of the required techniques would not be implemented in the available timeframe and the research hypothesis could not be adequately supported. The prototype demonstrates the principles of the system and its capabilities from such a subset. The majority of these results are of use for any data source or application.

1.3.3 Approach

In order to extract run-time data from the system for debugging, the JPDA library is used. JPDA allows for distributed debugging to take place on any platform running the Java Virtual Machine (JVM). The system parses data to the other features in the system for analysis and display.

The GUI is responsible for the display and collection of input from the user. This comprehensive GUI provides tools to view various levels of abstraction from system structure to the code level. It allows the user to select any variable for expression evaluation and trace through code to identify observed errors.

The system obtains the program structure to be represented by reverse engineering of the JPDA extract data. The object diagrams, which are similar to class diagrams but display instance information, are displayed dynamically, making them suitable for distributed debugging. The system displays diagrams following UML conventions, as UML is the most widely accepted notation for representing software architectures.

Debuggee programs, that is the program being debugged, may be extremely large, which implies the structural models will be very large. Focus + context visualization techniques incorporating selective aggregation and a fisheye lens allow a much larger portion of the program structure to be displayed on the screen while allowing the user access to detailed information for an area of interest.

The visual system includes a graph layout algorithm to increase the space efficiency of the diagram. The layered graph-layout algorithm preserves the hierarchical nature of UML object diagrams with layering based on inheritance relationships within the program. The visual transformation system applies the algorithm in two phases to preserve user context of the system where it already exists.

The debugger system highlights the execution path of programs on the structural view. This allows the user to see what the program is doing, rather than having to comprehend trace data.

In addition to the highlighting mentioned, a variety of visualization tools are included to increase the effectiveness of the system. This research selects color, layouts and motion based on the foundations of program visualization to achieve maximum effect [WAR00]. Focus + context is also included to allow a more complete view of the UML object diagrams presented by the system.

This research implements a prototype debugging system with the described features to evaluate the effectiveness of these techniques. Testing takes place to measure the effectiveness of these techniques in various environments.

1.4 Summary

The primary goal of this research is to develop more effective methodologies for debugging the JBI and other distributed systems. The researcher implements these techniques to meet the objectives in a Java based system allowing automatic data extraction during run-time. The data is reverse-engineered to present the user with UML object diagrams, providing the user with essential structural information on the program for debugging. The system also presents source code for the debugging process.

The research incorporates focus + context visualization techniques into the object diagram view. This allows the user to monitor a much larger portion of the program structure while maintaining access to detailed information for objects of interest. Selective aggregation is an ideal technique for use with UML object diagrams due to their hierarchical relationships. Other visualization techniques blend the view with global information and local detail.

The research performs testing with a prototype system to obtain evidence to support the effectiveness of these techniques. A variety of empirical and quantitative evidence supports the use of these techniques for debugging of large and distributed systems.

2 Literature Review

2.1 Introduction

This chapter discusses background topics relevant to distributed system visualization. The topics include debugging for single platforms and distributed systems, the JPDA, automatic bug detection algorithms, software visualization, and visualization in the context of debugging for distributed systems. A description of each of these topics follows along with its relationship to the research.

2.2 Data Sampling

This section explores various methods of extracting data from debuggee systems and factors that influence the appropriateness of these for use in distributed debugging for this research.

2.2.1 Monitor Types

A monitor is some system that allows the collection of data from a system while it is in operation [JAI91]. Three main types of monitors exist: hardware, software and hybrid (some combination of the two). The following describes each of these and their benefits and disadvantages in real-time monitoring with respect to two criteria. These criteria include the requirement to access data without significant changes to the system and the ability to gather information without altering system behavior or impacting performance [SIM90].

Pure hardware monitors often do not meet the requirements of application programmers. Their design and installation requires great expertise for the major modifications required and offers little or no flexibility. The data they measure is often too low level to be of use to application developers or testers, for example they may monitor a particular memory location rather than a variable. However hardware monitors have a distinct advantage over software and hybrid monitors. Hardware monitors do not interfere with the System Under Test (SUT). As such, the required data can be gathered with no effect to system behavior or performance.

Simmons explains that pure software monitors generally operate as a separate thread on the SUT recording the data required by the user [SIM90]. The data recorded can be rapidly modified should the user's requirements change throughout the course of the monitoring process, as they often do, particularly when the results are used for debugging. Significant interference to the results counteracted the flexibility offered by these systems. The monitor coexists in the execution environment of the SUT, sharing both the processor and memory with the monitored program. It can be very difficult to determine this effect, as many modern processors do not have constant execution times for a given instruction sequence [HEN96].

Hybrid tools combine the benefits of both of these approaches. They allow re-configuration of the extracted data and can present the data in a meaningful format, as can standard software monitors [SIM90]. Provided they have a hardware data-gathering component, they can also provide accurate data with minimal affect on the SUT.

2.2.1.1 Distributed Monitoring Systems

In distributed systems, a user cannot concurrently globally view or control all processes due to the lack of global time and state. As such, Lamport's Logical Clock is often used to provide timing information. This allows the order of operations to be determined but does not provide the time for an event as would a standard clock

[TAN02]. The same monitoring requirements still apply in this environment, i.e. low intrusion, flexibility and resolution. This can be difficult considering each monitor must send all data to a single location on the network.

For use in the JBI or any standard distributed system, hardware modification and the installation of separate data buses are impractical. As such, care must be taken to minimize the effects of the software monitor on the observed results.

2.2.2 Data Extraction

For an automated debugging tool, the method for data extraction from the debuggee system is an important factor. Appelbe discusses that many systems currently available for distributed debugging rely on time-consuming, cumbersome methods such as tracing which relies on the explicit calls to an output device to capture event data [APP93]. Topol hypothesizes that one of the main reasons for the limited use of visualization tools in distributed and parallel systems is the difficulty in gathering information [TOP94]. It is important for data extraction methods to be flexible, as developers rarely know what and when to display before debugging begins.

Topol explains that the most common methods for data extraction are recording of trace events and modification of middleware [TOP94]. Recording of trace events involves inserting print statements into the source code of the system either manually or via some automated aid. Obviously this is an intrusive procedure and very inflexible once program execution has commenced. Trace data produces large amounts of data that the system can save for later evaluation.

The other method Topol discusses, middleware modification, involves modifying the software protocol providing communication between the various systems on the network [TOP94]. Topol suggests this as an alternative to the intrusive nature of recording trace events as discussed above. Developers could add extra data to system messages allowing the creation of meaningful visualizations with minimal overhead. This method provides even larger amounts of data, as all events produce data. It is also possible to record this data for analysis later. However, this middleware modification still lacks complete flexibility and produces huge amounts of data that may not be required.

2.2.2.1 Java Platform Debug Architecture

The JPDA is a recent addition to the Java Software Development Kit (SDK) providing the capability for remote debugging of applications in the JVM as described in [SUN]. Since debugging occurs at the JVM and not at the application itself, debugging with JPDA can be performed by almost any two systems, a debugger and a debuggee, with the JVM installed, regardless of the operating systems or configuration. The goals of JPDA are to:

- Provide standard interfaces to simplify the creation and use of Java debugging tools, regardless of platform
- Describe the complete architecture for these interfaces allowing remote debugging
- Have a modular design

JPDA is comprised of two interfaces and a protocol. Figure 1 shows the integration of these components and is discussed in the following paragraphs:

- Java Virtual Machine Debugger Interface (JVMDI),
- Java Debug Interface (JDI), and

- Java Debug Wire Protocol (JDWP).

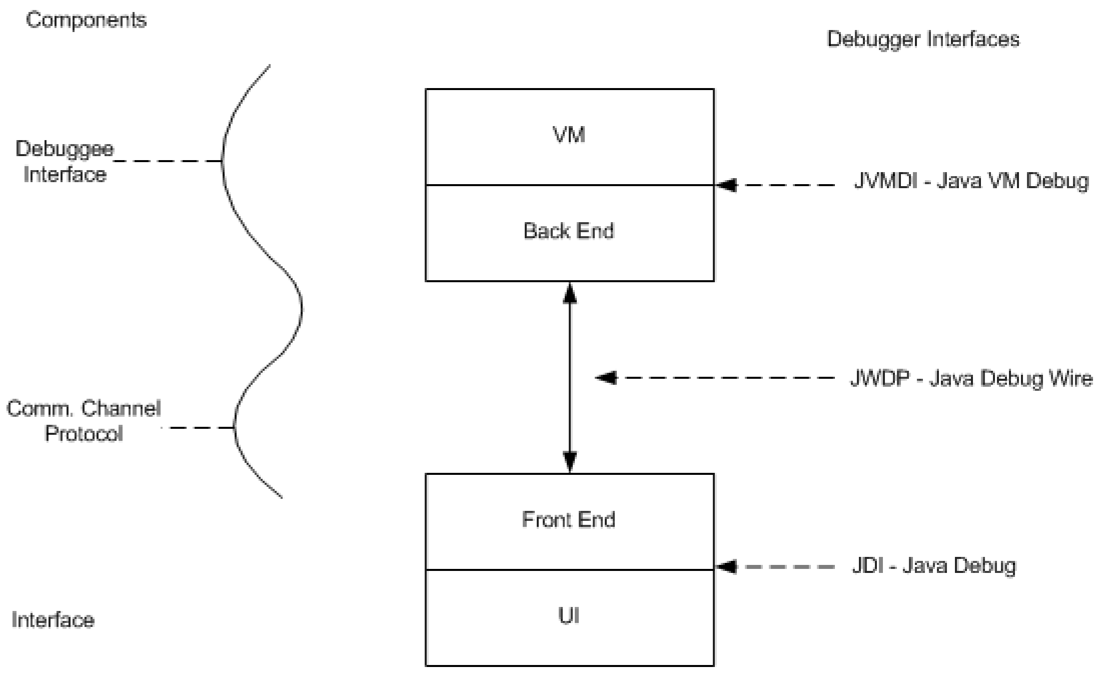


Figure 1. JPDA system overview.

JDI provides an interface to a remote view of the debugging process occurring in a JVM that may be located in another system. JDI is the most commonly used layer for access as it is the highest level and easiest to use.

JVMDI defines the interface for the JVM to allow debugging by debugger programs running in other JVMs. This is the source of all debugger specific information. It includes requests for information from the JVM, actions such as setting and removing breakpoints, and notification when the program counter reaches a breakpoint.

JDWP is the protocol that defines how two-way transfer of information should occur between the debuggee process and the debugger front-end. It does not provide low-level detail of the actual communication mechanisms; rather it defines the format of the data transfer between the debugger and debuggee.

The JPDA provides the capability to extract data in a platform-independent way from distributed systems for debugging. Use of the JPDA Application Programming Interface (API) means implementation of debugger connections and data extraction are relatively simple. The JPDA also reduces the effects of some of the problems caused by other data extraction techniques as described.

2.3 Debugging Issues

Software systems are becoming increasingly large and complicated as technology progresses and user requirements expand. This makes debugging difficult and is compounded by the fact that many developers are not familiar with the subject of the system they are creating. The following reviews some of the issues involved in the process of debugging and the essential tools for a debugger. Prior to this discussion on debugging, this paper introduces and defines bugs themselves.

2.3.1 Bug Background

Telles defines a bug as a problem with software that needs to be fixed [TEL01]. A bug may occur in the requirements, architecture, design, or implementations. Many errors in the expected operation of code can be worked around, however, the system should match with the users' requirements. Telles explains that software defects potentially become bugs, and on average 25 defects are contained in every 1000 lines of code.

Bugs can be introduced at every stage of the Software Life Cycle, but are generally a manifestation of the designer's lack of understanding of the domain. Kan found a correlation between the number of changes and enhancements made to a system and the number of defects [KAN95]. The study found that around 0.628 defects were introduced for each change/enhancement.

Telles classifies bugs into one of the following types [TEL01]:

- Requirement Bugs
- Design Bugs
- Implementation bugs
- Process Bugs
- Build Bugs
- Deployment Bugs
- Future Planning Bugs

As the bugs can be introduced at almost any stage, their effects may also vary greatly. Telles classifies the effects as one of the following [TEL01]:

- Memory or Resource Leaks
- Logic Errors
- Coding Errors
- Memory Overruns
- Loop Errors
- Conditional Errors
- Pointer Errors
- Allocation/De-allocation errors
- Multi-threaded errors
- Timing errors
- Distributed Application errors
- Storage
- Integration
- Conversion
- Hard-Coded Lengths/Sizes
- Versioning Bugs
- Reuse
- Boolean

The consequences of the bug depend on the severity of its effects on the system, ranging from minor annoyance to show-stoppers. Bugs affect companies through reduced morale, monetary expense and reputation.

2.3.2 Debugging

This section defines debugging and the activities that take place during debugging. Telles defines debugging as “the process of understanding the behavior of a system to facilitate the removal of bugs [TEL01]”. Fixing the symptoms alone does not solve the problem, and in fact may create new ones. Anecdotal evidence suggests that the probability of introducing a new error while attempting to fix another is between 15 and 50 percent [TEL01]. Fixing the symptom without correcting the cause is a result of a lack of understanding. Any tool that can increase understanding will go a long way towards reducing the number of bugs in a system.

There are two factors affecting understanding in the debugging process. First, one must understand how the system should operate based on the customer requirements. One must also understand the implementation of the system in order to recognize the differences in expected and actual behavior.

There is not a straightforward process to determine the location of a bug and correct it. Many other side issues affect the operation of a system particularly in distributed systems, where critical timing may exist and many systems are interacting with each other.

Bugs are reproducible though this may not always appear to be the case. Some bugs may be much harder to re-produce. It may take a complex series of events with critical timing to recreate them. Specific data inputs, environmental conditions or configuration may be only way to trigger others bugs.

Telles discusses several types of debugging methods ranging from the most methodical to virtual guessing [TEL01]. These are now examined.

2.3.2.1 Scientific Method

As with other engineering fields, the scientific method involves forming a hypothesis, gathering evidence to support or disprove it and continuing until the user can prove or disprove their hypothesis. This method is particularly useful when the problem is easily reproducible, for example, a particular input value results in an incorrect result.

2.3.2.2 Intuition

Intuition is a very common approach to debugging meaning the user “knows” where the problem is. It requires a thorough understanding of the code and where the bug is likely to be, based on the symptoms being displayed by the system.

2.3.2.3 Leap of Faith

In effect, a leap of faith is simply an educated guess. The developer examines some of the symptoms and jumps to conclusion without truly examining the behavior of the system. A leap of faith is more likely to lead one in the wrong direction.

2.3.2.4 Diagnostics

Diagnostic debugging, also known as advance strike debugging, involves predicting in advance the type and location of likely errors in the system and logging them for the programmer to correct later on in the analysis process.

Several approaches to debugging are discussed. When considered in relation to the goal of this research, i.e. developing a tool to assist in this process, then the tool must be developed to improve the most common cases first with consideration for as many methods as possible. The tool is not able to assist with predicting where bugs may occur before creation of the system (i.e. diagnostic debugging); however, it is envisaged that benefits are made to the other methods discussed.

2.3.2.5 The Debugging Process

Telles discusses guidelines for effective debugging of a system [TEL01]. As discussed, it may not always be an easy process to locate bugs, but a methodical approach may be of assistance. Telles guidelines are now reviewed in detail.

- Identify the problem. Find out if the problem can be reproduced and under what conditions. Try to find see what is happening so further information can be gathered.
- Determine if the problem is actually a bug. Make sure the bug is indeed a bug, there are cases where this may not be the case. If the bug is a problem, then determine why it does not match the specification or user requirements based on observations of the system, not examination of the code.
- Confirm what the program should be doing. Check the specification or updated user requirements to ensure that the correct operation of the system is known.
- Examine the program behavior. The operation of the system must be established to compare against correct operation of the system. This will require code analysis and is where tools can be particularly useful in narrowing down the search region.
- Gather information. Some ideas will be generated on what may be causing the error. To ensure that nothing is missed, all the relevant information

should be gathered. This may include collecting data or bug reports from users, examining log files generated by the system and personal observations by the tester. Symptoms must be examined to see if they are the actual bug or simply a side affect of its presence. Use test cases that remove redundant information from bug reports and narrow in on the cause. Similar problems should be examined in bug listings as the newly detected error may be caused by the same developer, algorithm or even reuse of code. Recent changes to the system should be carefully considered should the bug be a new occurrence. Changes should be well documented to assist in this process. Finally, environmental information and other external influences should be scrutinized to see if they too have changed. All these pieces of information will assist in determining the true cause of the bug in the system.

- With the information collected in these steps and knowledge of the correct and actual system operation, the developer should be able to come up with a hypothesis. The hypothesis should fit all the symptoms correctly. Should it not match all the evidence then further examination of the system may be required.
- The hypothesis should be evolved until the true cause of the bug is located and a solution found. Test cases should be executed to ensure correct program operation.

2.3.3 Debugging Tools

A variety of tools assist the debugging process by providing the users with all of the information they require. Effective tools transform the data to provide more meaning rather than just providing raw data. Test harnesses assist in limiting the scope of the tests to a likely region. Various debugging tools are able to narrow down the region of code under test. Logging and tracing are similar methods providing detailed information of the path of execution through the program. Telles recommends disabling the verbose output of these tools as soon as the likely region of the bug is determined [TEL01]. Without disabling the output, the volume of data provided may actually confuse the issue.

Once the likely region of the bug has been identified, mid-level debugging tools may be used to examine the problem more closely. This class of tools assumed the user has knowledge of the symptoms, likely location and a reasonable hypothesis of the cause of the bug. Examples of mid-level debugging tools include memory leak detection tools and, cross-indexing and usage tools. Telles discusses how memory leak detection tools are able to detect code that is likely to be causing a problem [TEL01]. Cross-indexing and usage tools are able to trace bottom-up through the code to find examples of where a function or a global variable is accessed. These tools are also able to find dead code within a program.

Debuggers allow the developer to stop the system in its execution at any point and examine the values of variables and in some cases step backwards through the code. These capabilities are highly desirable for any debugging tool.

2.3.4 Automatic Bug Detection Methods

The methods discussed have dealt with the broader subject of obtaining data to build for the user to develop an understanding of the system. Although these concepts are of great importance in distributed debugging, when combined with other tools, they can be far more effective.

Understanding of program operation significantly improves the probability of finding a bug and a good understanding reduces the time involved in this process. However, this method still requires the user alone to determine where the bug might be located based on their knowledge of the system and the programs' expected flow. For large systems, this process can be daunting. Three methods to combat this situation are discussed.

2.3.4.1 Invariant Based Detection

Hangal describes DIDUCE, an invariant based bug detection system allowing automatic detection of errors and their causes in Java based systems [HAN02]. The system formulates hypotheses of invariants in program behavior. The invariants are initially created at a very strict level and relaxed as program flow proceeds and new acceptable behaviors are detected. Should a significant error pass through the system, the invariant would have to be relaxed drastically to accommodate it. In this way, bugs can be rapidly detected.

DIDUCE has been successfully used on several Java programs of significant size and complexity. It has been able to detect algorithm and input errors in addition to finding problems in the interface between program modules [HAN02]. The system has also been able to highlight rare cases often forgotten or inappropriately dealt with. This type of detection method is best used when the debuggee system requires little human interaction.

2.3.4.2 Visualization of Test Results

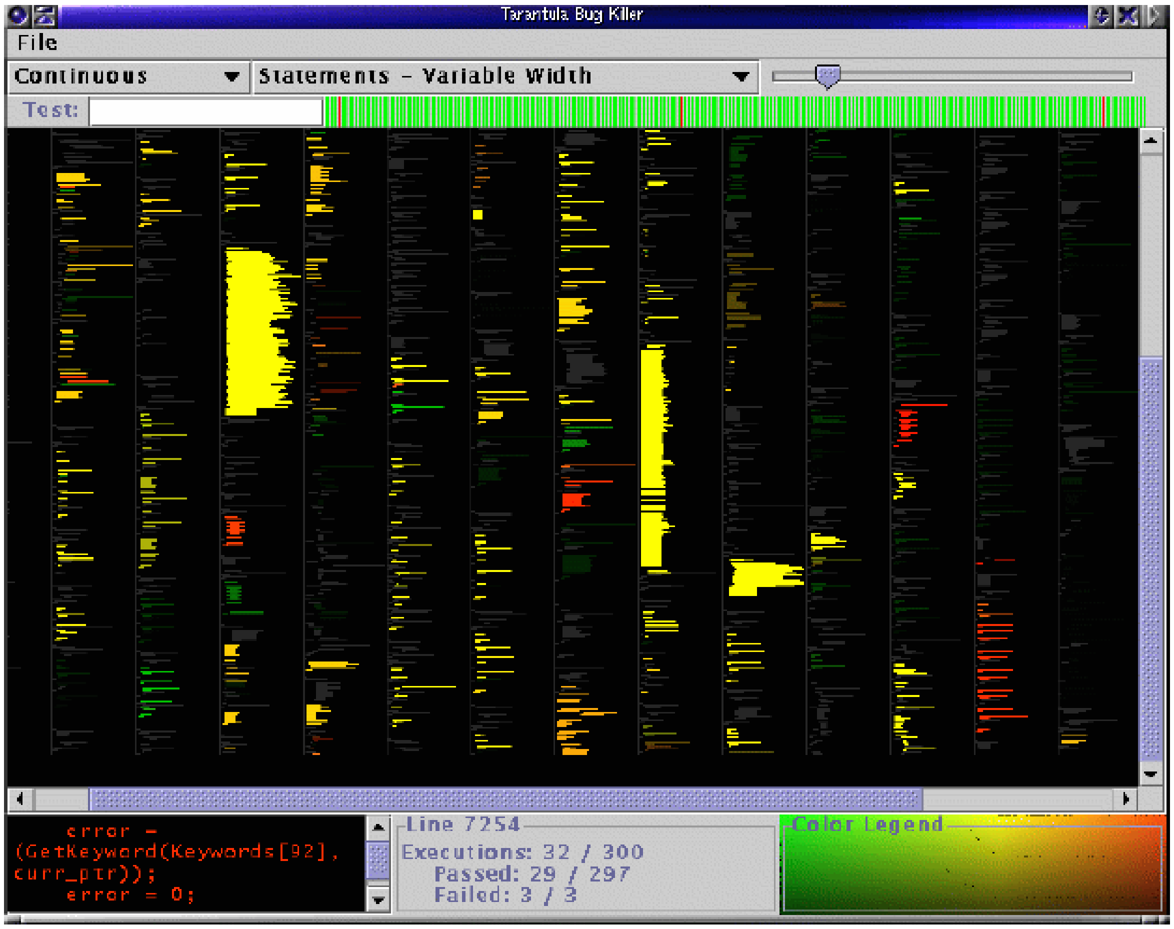
Another approach, developed by Jones, relies heavily on existing test cases and results to locate errors [JON02]. Such a system analyzes the execution of the program and determines the probability of a bug occurring for each line of code. This is easy to compute, knowing the lines of code each test case executes and whether or not the test case is successful. In this manner, the flawed region can be easily located, particularly if it lies within a conditional branch.

Tarantula, the system Jones discussed, displays the results of this analysis using visualization as in Figure 2. Colors highlight regions of the program that are likely to contain flaws [JON02]. The user can select these regions to analyze the program at the code level. Typically, this system requires the execution and analysis of thousands of test cases for meaningful results. As a result, it is less automated than the approach discussed previously.

2.3.4.3 Indirect Detection Methods

Other methods focus on detection of bugs through visualization of program execution parameters. Although they do not usually directly locate an erroneous line of code, they can go a long way towards locating the bug. One example of this is the Software Visualization Supporting Space (SVSS) as described by Amari [AMA99]. The trace-data comparison view in this system visualizes module interaction against time (since commencement of program). Absence of interaction between models and incorrect interactions can reveal flaws in the program.

One of the main benefits of this approach is that is generally far easier to implement, as less analysis is performed by the software. It is then able to take advantage of pre-cognitive processing by the viewer. However to make judgments requires knowledge of how the execution should look when performing correctly.



```
    error =
(GetKeyword(Keywords[92],
curr_ptr));
    error = 0;
```

Line 7254
Executions: 32 / 300
Passed: 29 / 297
Failed: 3 / 3

Figure 2. Tarantula source analysis display.

SVSS also contains a view to display specified data in the program. By displaying data of interest, the user can closely monitor likely fault regions in the program. For example, the program may display results of all those modules that read or alter the value of a variable and the order of these operations. This is particularly useful in situations where the program handles exceptions or invokes warning messages, as the state and location of the error causing code can be seen without any further effort. Figure 3 shows a screenshot of SVSS where functions highlighted in black directly call error routines in the system.

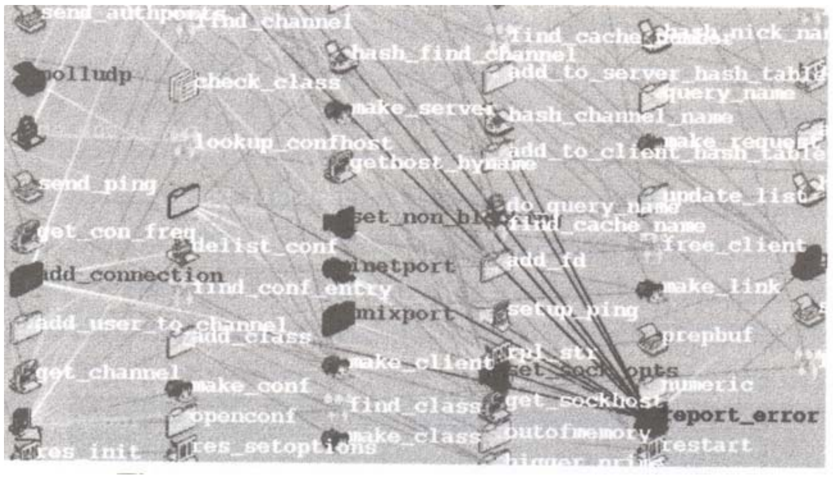


Figure 3. SVSS display highlights functions that call error routines.

2.4 Visualization

In recent times visualization with the aid of computers has greatly added to a human's ability for problem solving. In short, visualization greatly increases the bandwidth humans can accept data with and rapidly process, finding visual oddities with the image.

The term visualization has evolved along with the capabilities of our technology. An accepted definition for the term is a graphical representation of data or concepts as an external construct supporting decision-making [WAR00].

Visualization helps us to work more efficiently in several ways. One of these is its effect on our usable memory. Miller describes how human working memory is only capable of handling five to nine chunks of information at a time [MIL56]. Ware defines a chunk in the field of cognitive psychology as an important unit of stored information, often the aggregate of other pieces of information [WAR00]. A chunk can be almost anything, an object, an attribute or an image. Most importantly, visualization allows the user to leave the image of the chunk on the display rather than storing it in their working memory. In addition, a user can detect interesting data without cognitive effort by observing changes in patterns and other visual attributes.

Ware explains that due to pre-attentive processing, certain parts of an image can tend to jump out at us [WAR00]. This processing is similar to a filter. Pre-attentive processing determines the visual objects to pass up for attentive processing. This is probably the most important capability to possess for data visualization - the ability to filter millions of data points without attentively processing each of them. Various visual attributes take advantage of pre-attentive processing; these include:

- Color,
- Shading,
- Patterns
- Movement, and
- Brightness

Ware discusses the details of the relative affect of each of these attributes and those that are most effective [WAR00].

The importance of memory and the power of pre-attentive processing from data visualization have been discussed. How the visualization is used can also affect its worth. Cultural differences may alter the perception of a symbol to different people. In visualization design, one must be careful to avoid unknowingly triggering an optical illusion. Ware discusses several types of optical illusions and how to overcome them [WAR00]. Card mentions the problems associated with the comparison of visual objects [CAR99]. These problems may prevent the user from gaining even a qualitative insight from a picture. Position and shape are the two main factors to consider to avoid illusions.

2.4.1 Program Visualization

Having considered the benefits of data visualization and methods to optimize it, this research evaluates visualization techniques for program execution. Visualization techniques are also discussed for distributed systems, debugging and distributed debugging.

Stasko advises that an effective software visualization environment builds on the text based techniques generally used, with displays that make better use of the human-computer interface [STA98]. To be more valuable than these text-based methods, the visualizations must filter information for relevance (by user manipulation) and the user must easily understand them.

The user rarely knows of erroneous behavior from the observed system in advance. As such, the system must be flexible to capture this dynamic data. Run-time data must be displayed dynamically making it more difficult to present than static data. With this in mind, Stasko developed the following principles for dynamic software display [STA98].

Animation – Animation has the ability to display temporal relationships. Animation can be constructed by modifying size, shape, position or the appearance of an object.

Metaphors – metaphors are symbols that represent objects in a way which reduces the cognitive load on the user. The system should minimize extra learning required to recognize these symbols.

Interconnection – Interconnection is used to represent relationships between components and their patterns of behavior.

Interaction – Visualizations are rarely effective without the user controlling what to display, how, and when. The controls should be as simple as possible.

Elaborate systems such as Stasko's Tango perform algorithm animation. In Tango, the user specifies the animation to construct from generic transformations [STA90]. This research aims to minimize the complexity of animation by limiting the number of behaviors that are animated and selecting simple animation types.

A variety of common techniques has been developed for program visualization. Stasko reviews a variety of methods, each having its own strengths [STA98]. An important issue with program visualization is the tradeoff between performance of the visualization and its ease of use.

2.4.1.1 The Unified Modeling Language (UML)

UML provides diagrams to model static software structure and different aspects of dynamic system behavior [OMG00]. The software industry is widely aware of UML's symbols and associated semantics. To accept any visualization solution for software systems, the software community requires the use of UML symbology and semantics. UML has a variety of diagrams relevant to software engineering including class, use case, sequence, state and deployment diagrams.

The class diagram is the most widely used of all UML diagrams [COO99]. A class diagram includes rectangular nodes depicting classes with annotated lines between these nodes to indicate the relationships between classes. Rectangular nodes depicting classes are annotated with textual labels to identify the class and its associated state variables and behaviors. Software systems often consist of hundreds or thousands of unique classes. Analysis and comprehension of such a software system requires both a high-level overview of the system structure (consisting of numerous classes and the relationships among them) and a detailed examination of the characteristics of individual classes or small subsets of classes. Software visualization techniques should provide access to both high level and detailed views.

2.4.1.2 Program Visualization Evaluation Techniques

Visualization of program structure, control flow and data has long been a part of software development. Examples include flow charts, class diagrams, state-charts, pretty printing of code, and algorithm animation. To evaluate program visualization techniques, Roman and Cox propose a taxonomy consisting of five criteria – scope, abstraction, specification method, interface and presentation [ROM93]. Price et al identify additional criteria to include fidelity and invasiveness [PRI97].

More powerful visualization techniques present a broad *scope*, covering many aspects of a program such as code, data state, control state, and behavior. In presenting this scope, these techniques should provide many views with multiple levels of *abstraction* from high-level design to code. Access to multiple levels of abstraction gives the user the ability to see overview for context while having access to lower level information necessary for detailed problem solving. The visualization system needs *specification methods* that require minimal effort from the user while providing sufficient flexibility for the viewer to customize the content they explore. The *interface* for interacting with the visual presentation should be intuitive and easy to understand and use. In general, direct manipulation interfaces tend to be more intuitive than interaction through controls. *Presentation* semantics must be sufficiently abstract and powerful to reduce the cognitive load on the viewer. It is especially important to capture the complex relationships between various program aspects. Specification and presentation of program information must minimize *invasiveness* to the program while maintaining *fidelity*. The visualization should not alter the behavior of the program or use misleading semantics that lead to incorrect interpretation.

Debuggers such as those in Borland JBuilder illustrate the low end of program visualization techniques [BOR]. These debuggers provide little abstraction and minimalist interface capabilities, that is, debugging takes place with a GUI but the data itself has not undergone any transformation. In general, users are able to interact with textual representations of code and data state using simple text, menu, or button commands. Users have very limited capability to specify the desired information and visual presentation. Presentation semantics are no more than those provided by the underlying code.

2.4.2 Visualization of Distributed Systems for Debugging

Researchers must consider a variety of complications when designing a system for the visualization of distributed or concurrent programs. These systems are by their nature, larger and vastly more complex producing large amounts of data in comparison to a standard sequential program. Stasko surmises that knowledge of the nature of the interaction between components and the timing of their interactions becomes the broad goal of distributed visualization systems [STA98].

Wong discusses a generic visualization framework for debugging object-based distributed programs [WON01]. This proposed system relies on detecting errors in a distributed program based on an agent's location within the system. It assumes the agents are all operating correctly as standalone systems.

Three processes must take place for effective program visualization, particularly in distributed systems. These processes are data collection, analysis and display. The following sections analyze each of these processes in further detail.

2.4.2.1 Data Collection

Data collection in distributed systems results in a further complication. Unlike sequential systems, multiple breakpoints must be set on different systems in multiple locations. The system must also deal with communications in transit at the time and a variety of other shared resources.

Stasko discusses the types of interesting events that may be suitable for collection via instrumentation of the program and the levels the collection can take place [STA98]. For distributed and concurrent systems, interesting events at the operating system level may include message sends and receives, process creation, scheduling, page faults, context switching, memory access, and system calls. At the run-time level, the state of queues, lock actions, critical points and procedure calls and returns may be of use. Finally, at the application level, data from any data type within the program may be of interest and it should all be obtainable. Data at the application level is of a higher-level and more abstract. Data from all of these layers is of interest when debugging a system.

Monitoring of distributed systems generates large amounts of data at a rapid rate. As such, the collection should be as restrictive as possible, yet provide all that is required. The use of this data in real-time monitoring introduces further complications. The data and events happen so rapidly that a human is unable to monitor them effectively. One possible solution for this problem is the recording of events. Recording creates large data files that allow the user to replay the data and trace the error. A second option actually stops or slows down the execution of the program, as is often the case with debugging.

2.4.2.2 Analysis

Analysis involves the processing of collected data into a different format or into a more abstract representation that is more suitable for display. Analysis components may calculate statistics for performance analysis of the system to provide an overview of operation. Stasko describes that for any real-time system, CPU time constraints limit the amount of analysis [STA98]. With improved data analysis, the resolution of the displayed information increases and most likely becomes more effective. However, the delay between updates to the display will increase with this extra processing. This is yet another tradeoff to consider in visualization system design.

Analysis in distributed systems often involves detection of non-determinism between processes. Such analysis is NP-hard or worse and may be more easily detected from the display of event-trace graphs [STA98].

2.4.2.3 Display

The goals for displaying distributed programs are the same as those for single-platform systems. That is, the display should highlight data using appearance, shape, position and movement to aid the user. Stasko also states that extra difficulty arises from the scale of these systems and event ordering issues [STA98].

The most important events to convey are the interactions between processes. The display must show this accurately and preferably with some aggregation of events to remove complexity from the display so as not to confuse the user [JER97]. Displays are most effective when they match with the mental image the user already has of the system and are generally more abstract as they occur at a higher level.

In addition to displaying the behavior of the system, it is important to display the abstract view with the source code. It is also preferable to link the code with the matching region in the abstract view through highlighting, allowing the user to quickly locate and correct an error. [JER97]

Researchers use several types of displays to represent distributed system behavior. These displays are reviewed here.

2.4.2.3.1 Graphs

Graphs are often used in system displays where the vertices represent some module in the system and the arcs represent communication or a temporal order [KRA97]. Figure 4 shows a graph view from the Gthreads package. This view shows the creation of a new thread at the fork and traces the execution from function to function [KRA97]. Animation can effectively show the causality of events in the system and those modules that are involved. This is particularly useful for debugging and can be useful for detecting unexpected behavior sequences or modules within the program.

2.4.2.3.2 Circular display

Bowman describes how Conch arranges processes around the outside of a circular display [BOW94]. Messages, shown as small colored discs, move from the originating process to the receiving process. Undelivered messages remain in the middle of the circle, allowing the user to identify them easily as shown in Figure 5.

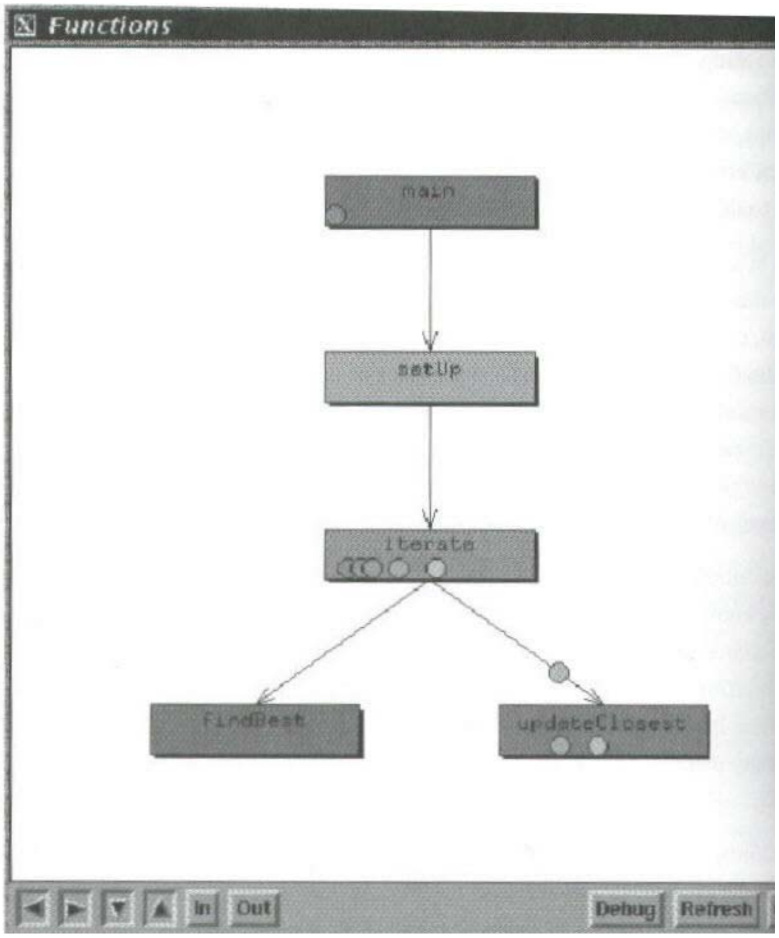


Figure 4. Graph view from the Gthreads package.

2.4.2.3.3 Time-process graph

Many visual debugging systems provide a time-process display. The display shows the time domain on one axis with individual processes drawn on the other axis as shown in Figure 6 [KRA97]. Time-process graphs show the synchronized interaction amongst processes as intersections between these lines. Jerding discusses the benefits of collapsing hierarchical processes to reduce clutter from the view or allowing examination in more detail if required [JER97].

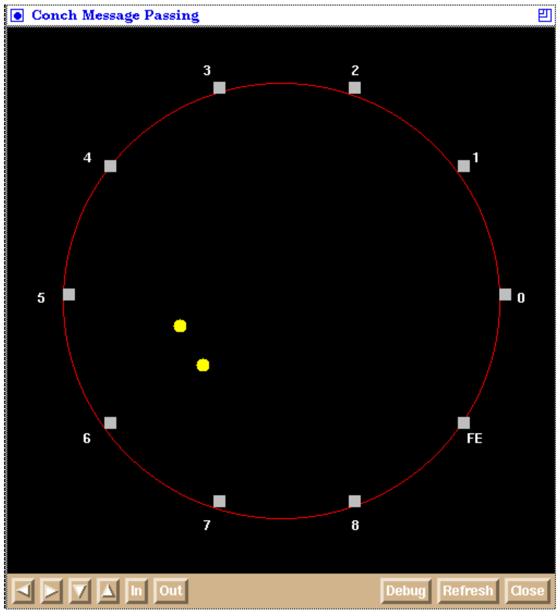


Figure 5. Conch Process communication display

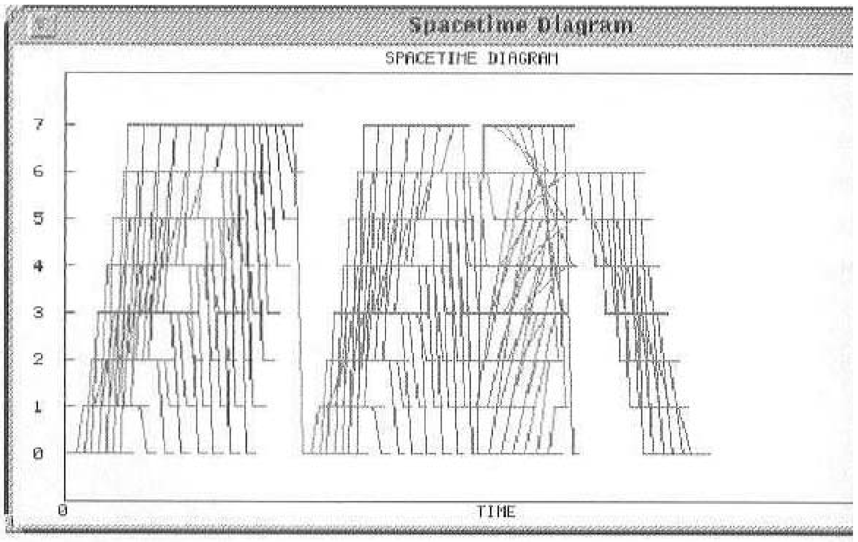


Figure 6. Time-process communication graph from ParaGraph.

2.4.3 Visualization Methods

As discussed previously, there is a massive amount of information available with large and distributed systems. Visualization methods exist that can make these large amounts of data far easier to deal with for the user. This section focuses on those techniques showing a complete view of the data while providing detail for an area of interest. There are two main methods that do this with the display. Focus + context shows detail within the existing display, and overview + detail maintains separate areas for displaying the different levels of detail.

2.4.3.1 Overview + Detail

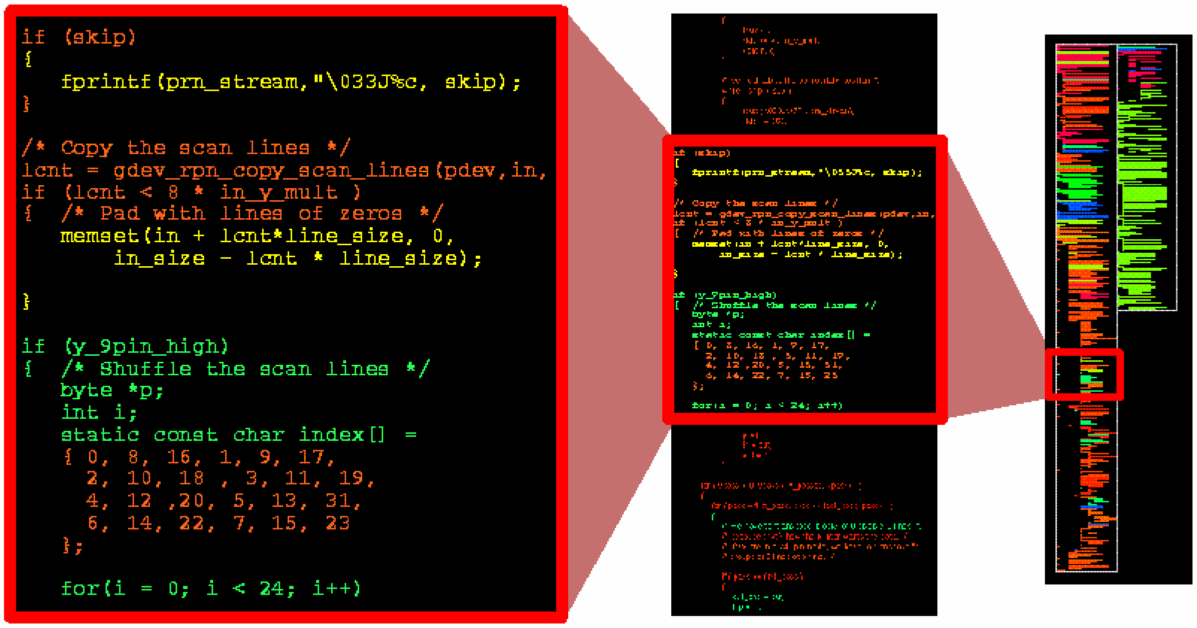
Overview + detail presents global location with detail shown for an area of interest. Card describes the advantages of this technique. The overview reduces search within the data display and improves the ability to detect patterns [CAR99]. The display of the overview also allows the user to decide more easily their next move. The display of the detail on the other hand allows rapid access to meaningful data. This combination of the two views allows the user to track a region of interest in the global display while having detailed information to use.

This method is generally implemented with some highlighting of the global view to indicate the current region of detailed display. The detailed information is then presented elsewhere. The user then easily tracks updates to the position in the global context. Figure 7 shows an example of this technique applied to code, the highlighting is shown on each higher-level view [CAR99]. Shneiderman advises that this zoomed detail view should have an effective zoom factor of between 3 and 30. A visualization should present intermediate views if the view factor is greater than 30 [SHN98].

Card states that the detailed view can be presented either in a different location on the screen (space multiplexing) or at a different time (time multiplexing) [CAR99]. There are obviously tradeoffs associated with each approach and the ratios involved in the multiplexing between the two views.

The detailed view may be a scaled view of the overview or it may use a different representation to present more information with extra clarity. For example, a user will not easily recognize a photograph of a hospital on a map, particularly if they are not familiar with the pictured hospital. They will recognize a symbol of a hospital such as a red cross far more easily.

User control of the views becomes an important issue in the design of overview + detail visualizations. Zoom-and-replace is a technique used with overview + detail where a mouse-click by the user on a location in the global display results in the display of the selected location in detail.



```
if (skip)
{
   fprintf(prn_stream,"\033J%c, skip);
}

/* Copy the scan lines */
lcnt = gdev_rpn_copy_scan_lines(pdev,in,
if (lcnt < 8 * in_y_mult )
{  /* Pad with lines of zeros */
   memset(in + lcnt*line_size, 0,
       in_size - lcnt * line_size);

}

if (y_9pin_high)
{  /* Shuffle the scan lines */
   byte *p;
   int i;
   static const char index[] =
   { 0, 8, 16, 1, 9, 17,
     2, 10, 18 , 3, 11, 19,
     4, 12 ,20, 5, 13, 31,
     6, 14, 22, 7, 15, 23
   };

   for(i = 0; i < 24; i++)
```

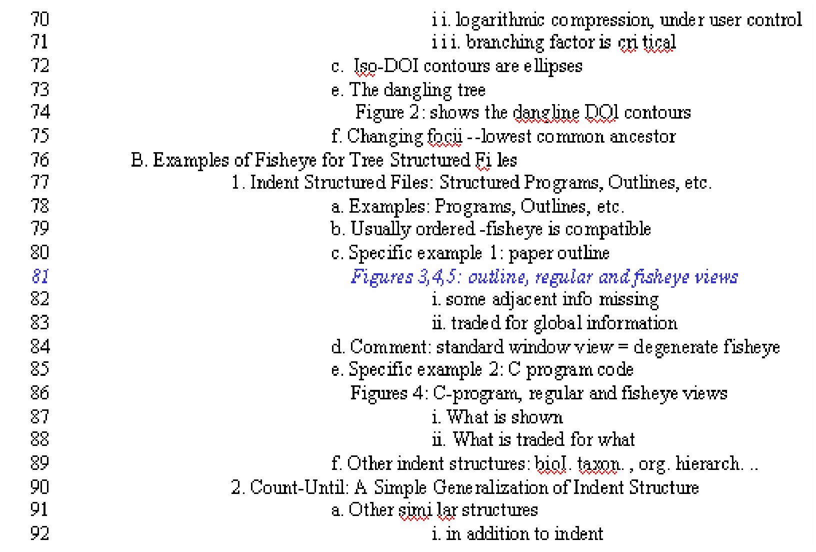
Figure 7. Overview + detail technique with intermediate view (from SeeSoft).

With use of a model, the user develops some contextual awareness. During the transition to an alternate display, it is highly desirable to maintain this context. Along these lines, Wickens develops the concept of visual momentum to create a set of user-interface design principles [WIC92]. One of these principles suggests that to preserve context, smooth transitions should be used so that the relative position of each element may be tracked.

2.4.3.2 Focus + Context

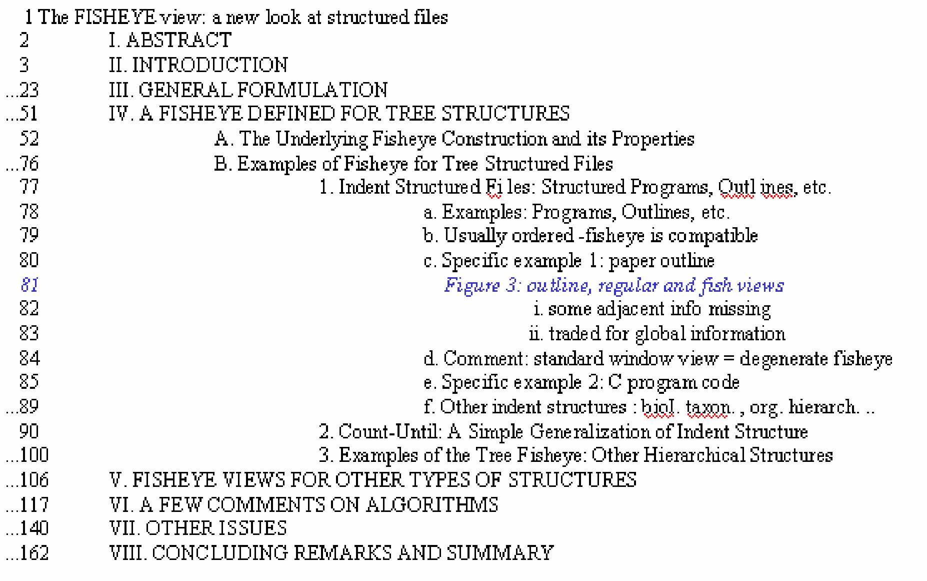
Card also reviews the focus + context technique. The concept behind it is based on three premises [CAR99]. The first is that the user requires both the overall picture and a detail view. The second premise states that there may be different requirements for the information in the detail view than in the overall display. The last premise proposes that the visualization combines this information into a single dynamic view. Figure 8 shows a standard flat view of text from a chapter in a book displaying sequential headings. Figure 9 shows a fisheye view of the text demonstrating Focus + Context, which shows some localized headings with context provided by major headings for the entire chapter.

Bertin explains that it is beneficial to combine the two views because when information is separated into two regions, visual search and working memory limitations result in reduced performance [BER77]. A second reason to combine the two views is identified by Furnas during research into fisheye views [FUR81]. It showed that the user's attention drops off away from the areas of detail. Various methods take advantage of this observation, relating the level of detail to the apparent interest in the region by the user. This improves the space-time efficiency of the user, a limited resource. This efficiency is increased as more information that is useful is displayed per unit area and the average time to find an item of interest is reduced as it is more likely to be already displayed on the screen. Card further explains how focus + context uses these principles to reduce the cost of accessing information which in turn results in increased cognition [CAR99].



```
70                              i i. logarithmic compression, under user control
71                              i i i. branching factor is cri tical
72                         c. Iso-DOI contours are e llipses
73                         e. The dangling tree
74                            Figure 2: shows the dangline DOl contours
75                         f. Changing focii --lowest common ancestor
76         B. Examples of Fisheye for Tree Structured Fi les
77                1. Indent Structured Files: Structured Programs, Outlines, etc.
78                         a. Examples: Programs, Outlines, etc.
79                         b. Usually ordered -fisheye is compatible
80                         c. Specific example 1: paper outline
81                            Figures 3,4,5: outline, regular and fisheye views
82                              i. some adjacent info missing
83                              ii. traded for global information
84                         d. Comment: standard window view = degenerate fisheye
85                         e. Specific example 2: C program code
86                            Figures 4: C-program, regular and fisheye views
87                              i. What is shown
88                              ii. What is traded for what
89                         f. Other indent structures: biol. taxon., org. hierarch. ..
90                2. Count-Until: A Simple Generalization of Indent Structure
91                         a. Other simi lar structures
92                              i. in addition to indent
```

Figure 8. Flat view of chapter.



```
 1 The FISHEYE view: a new look at structured files
 2         I. ABSTRACT
 3         II. INTRODUCTION
...23      III. GENERAL FORMULATION
...51      IV. A FISHEYE DEFINED FOR TREE STRUCTURES
 52                  A. The Underlying Fisheye Construction and its Properties
...76                B. Examples of Fisheye for Tree Structured Files
 77                           1. Indent Structured Fi les: Structured Programs, Outl ines, etc.
 78                                    a. Examples: Programs, Outlines, etc.
 79                                    b. Usually ordered -fisheye is compatible
 80                                    c. Specific example 1: paper outline
 81                                       Figure 3: outline, regular and fish views
 82                                              i. some adjacent info missing
 83                                              ii. traded for global information
 84                                    d. Comment: standard window view = degenerate fisheye
 85                                    e. Specific example 2: C program code
...89                                  f. Other indent structures : biol. taxon., org. hierarch. ..
 90                           2. Count-Until: A Simple Generalization of Indent Structure
...100                        3. Examples of the Tree Fisheye: Other Hierarchical Structures
...106     V. FISHEYE VIEWS FOR OTHER TYPES OF STRUCTURES
...117     VI. A FEW COMMENTS ON ALGORITHMS
...140     VII. OTHER ISSUES
...162     VIII. CONCLUDING REMARKS AND SUMMARY
```

Figure 9. Furnas fisheye view of text from Figure 8.

Systems can implement focus + context with a variety of methods that provide selective reduction of the presented information. These methods include filtering out extraneous information, selective aggregation of related information, micro-macro readings, highlighting, and distortion [CAR99].

Of particular interest for graph structures is the concept of selective aggregation. This method hides aggregate elements within another component. To achieve a focus + context view, hierarchical structures collapse away from the user's focus. When the user brings content into focus, the hierarchy expands revealing the aggregate relationships within. UML has aggregation and inheritance relationships that are ideal for this technique.

Other transforms may be required to blend the aggregate view with the global view. Several fisheye lenses are available for this purpose as discussed by Leung [LEU94]. In order to preserve the appearance of links and location context in UML, a lens that minimizes distortion is best for this application. As such, a step-wise function similar to that used in the Furnas Fisheye [FUR81] is more suitable than nonlinear or polar transforms.

Köth and Minas apply focus + context techniques to UML models in DIAGEN [KOT01]. DIAGEN is designed to provide adjustable detail levels for editing large diagrams. When used for UML diagrams, DIAGEN uses selective aggregation to hide the contents of packages and other components to reduce the amount of detailed information displayed. While package aggregation can improve navigation and access to system level structures, it does not address the structures within packages that can still be considerably large and difficult to navigate.

2.5 Graph Layouts

Just as the above-mentioned visualization techniques can assist the user in comprehension of an image, graph layout algorithms may have a drastic effect on the clarity and space efficiency of a graph. Graphs are often used to model structures such as software, where the nodes or vertices are entities within a software model and edges in the graph represent relationships [BAT99].

The aesthetics of a graph describe its graphical properties. Commonly adopted aesthetics include crossings, area, total or uniform edge length, total or uniform bends, angular resolution, aspect ratio and symmetry [BAT99].

Constraints describe how to lay out sub-graphs within the main graph. For example, a diagram representing a sequential model may show a number of nodes in a left-right sequence. Constraints control the layout in order to meet the expectations of users (conventions) or standards of a particular system. Examples of constraints include center, external, cluster, left-right sequence and shape [BAT99].

For any graph, there is no absolute best solution because aesthetics often conflict. Even if the chosen aesthetics do not conflict, it is often impossible or certainly computationally expensive to optimize them all. As such, precedence in aesthetics is often a necessary compromise for any graph layout algorithm [BAT99].

The algorithm to layout a UML class diagram should preserve the hierarchical nature of the diagram. A variety of tools for displaying class diagrams present generalization hierarchies down the vertical axis [RAT], [AUB]. Several layout techniques are analyzed with the aim of implementing this hierarchical transformation. The best algorithm for this purpose is a layered drawing of the digraph [BAT99]. A layered approach allows for a hierarchical presentation with vertices arranged in horizontal layers as shown in the example in Figure 10. One can apply this approach to any directed graph. By considering the UML diagram to be undirected for the purpose of layout, one can reduce complexity in the algorithm. The layered method consists of the following steps:

1. Layer Assignment: Assign vertices to horizontal layers, this determines their y-coordinate.
2. Crossing reduction: Order the vertices within each layer to minimize edge crossings.
3. Horizontal coordinate assignment: determine the x-coordinate for each vertex.

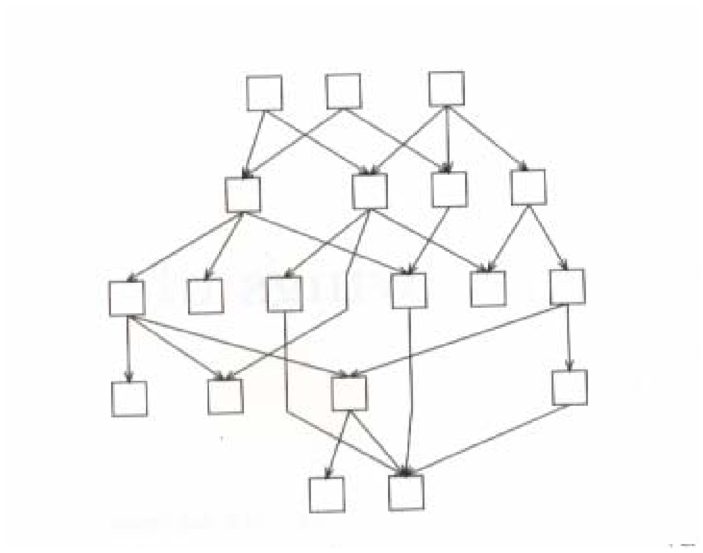


Figure 10. Example of diagram with layered layout.

The first step, layer assignment, requires a process to determine the layer for each vertex. For UML, a layer difference exists across each generalization relationship, with the top most vertex being the most generalized class. In the simple case where each vertex is the same size, the y-coordinate of each layer is determined by adding a suitable gap to the last layer. The standard layered approach requires that the digraph be compact in both dimensions and that the gaps between vertices are fixed.

For the second step, Di Battista describes the insertion of dummy nodes to ensure that a relation does not directly cross more than one layer. This ensures minimization of edge crossing [BAT99]. When the vertices are of variable size, the insertion of dummy nodes will not prevent edge crossings in the graph.

The third step requires the positioning of each vertex on a layer. The choice of algorithm for this step is dependent on user requirements.

2.6 User Interfaces

User interactivity and control is an important subject to consider in software design. The user interface has a large affect on whether a program has continued use, even if the functionality itself is acceptable. As such, the system implemented in this research should strive to meet the goals required for an effective interface. With this in mind, Shneiderman discusses some goals for effective GUI design [SHN98].

2.6.1 The Golden Rules for GUI Design [SHN98]

- Strive for consistency – use consistent sequences of actions as well as identical terminology for prompts.

- Enable frequent users to use shortcuts – Hotkeys reduce the time taken to initiate an action for frequent users.

- Offer informative feedback – System responses to user requests should be rapid and helpful to the user.

- Design dialogs to yield closure – Group sequences of actions with an informative and meaningful end to that group.

- Offer error prevention and simple error handling – design the system so the user cannot make a simple error and offer simple instructions to recover.

- Permit easy reversal of actions – this allows the user to try something without the fear of errors destroying their work.

- Support the internal locus of control – make the user the initiator of any system response so they feel they are in control.

- Reduce short-term memory load – keep displays simple with online help to information.

2.7 Summary

This chapter discusses some important background information relevant to visualization and debugging for distributed systems. Data extraction techniques, including hardware, software and hybrid monitors are covered. The chapter discusses a variety of debugging issues including automatic bug detection systems.

Visualization techniques greatly assist other chosen methods in meeting research goals. Several techniques are discussed for program execution visualization, including monitoring of distributed systems. This chapter introduces visualization techniques for debugging of standalone and distributed systems. In addition, it examines methods for display of global context in conjunction with detail information. These techniques are essential for dealing with large program structures. Finally, the research presents some guidelines for GUI design to ensure effective computer-human interaction.

3 System Design

3.1 Introduction

Software engineers typically use UML to communicate software analysis and design models, but once these models are implemented, the UML is seldom referenced. Only recently have researchers begun exploring languages such as UML for execution analysis. Examples include Mehner's extensions to UML to incorporate execution semantics of concurrent programs [MEH00]. The disconnect between design and implementation is made worse by the lack of automated support for testing and debugging with UML. Debugging any system requires structural knowledge of the software and detailed information about components (down to the source code level). Traditional debugging involves the user creating a mental image of the structure and execution path based on source code. As Miller discusses, the 7 ± 2 rule makes it very difficult for humans to construct large mental models [MIL56]. To alleviate this problem, this research proposes to create a visual execution model of the software for the user.

3.2 Objectives

The overarching goal of this research is to develop techniques to simplify distributed debugging. As discussed in Chapter 2, the use of visualization techniques can assist with the presentation of large amounts of information. Visual techniques may allow access to more relevant data in less time with less cognitive processing and can make the transfer between human and computer more effective. The goal is to provide a visual presentation that facilitates system evaluation using high-level design representations rather than stepping through lines of code. The hypothesis developed from this goal is that providing the user with a visually enhanced debugging system is more effective than a standard debugger for large and distributed systems. In order to assess the success of the hypothesis, a prototype system is created and tests run. The following sections present the visual and system objectives.

3.2.1 Visual

As discussed above, the intention of the visualizations is to enhance the debugging experience for the user. For effective debugging, it is essential for users to have access to various levels of abstraction. The system should present both high-level views and detailed information to allow precise debugging.

Paige identifies nine factors to consider when designing and evaluating modeling languages—simplicity, uniqueness, consistency, seamlessness, reversibility, scalability, supportability, reliability, and space economy [PAI00]. One could argue the degree to which UML satisfies many of these factors. However, this research only considers space economy. Space economy requires that "models should take up as little space on the printed page as possible [PAI00]." UML does not effectively address space economy, leaving lots of unused space within and between components.

UML's inefficient use of space results in models that cover many pages for large systems. Navigating through many pages significantly increases the time to access components of interest. In addition, by spreading the model over multiple pages, it is not possible to simultaneously view high-level system structure and individual component details. One goal of this work is to maintain the symbology and semantics of UML while improving its space efficiency to accommodate rapid access to both high level and detailed system information.

The research summarizes the visual requirements of the system as follows:

- The user should have views ranging from source code to high-level architectural views,
- The model should be represented in a familiar manner,
- The system should preserve user context,
- The display should be dynamic to match the changing run-time nature of software, and
- The system should provide improved space efficiency to reduce search while presenting the user with detail for a region of interest.

3.2.2 Design

This research achieves the above visual objectives in a prototype system to facilitate evaluation of their effectiveness. This section addresses the functional, performance and user requirements for this system to be effective. The system must be user-friendly and functional. Software engineering principles, including the use of design patterns, are considered to make the tool more easily adapted to meet future requirements. As many of the actions and their associated algorithms are computationally intensive, there is also consideration of efficient data storage and processing. The prototype system should have:

- The ability to monitor or spawn processes for debugging in a distributed environment,
- The capability to automatically reverse engineer systems,
- The ability to display multiple systems over socket connections for debugging,
- Unchanged system behavior of monitored programs,
- Minimal effect on system performance,
- Standard debugging controls,
- Well designed GUI with consideration of Shneiderman’s rules [SHN98],

- Low CPU load and memory usage for the monitoring system, and
- A design adhering to the principles of software engineering wherever possible to ensure it adapts for future requirements.

3.3 Experimental Techniques

To test the hypothesis that a visual debugger with the capabilities described is more effective than a standard debugger for large and distributed systems requires testing in the intended environment. As such, a prototype system is implemented to allow this comparison to take place. In developing a prototype system, greater testing and refinement of the methods can take place leading to a more thorough evaluation of the hypothesis.

Most objective methods require months of test design, special facilities, and user trials on many subjects to provide quantitative results [USA]. Examples of these objective methods include the performance measurement technique and retrospective testing. Alternatively, users or expert analysts may compare the system with a previous system in a more subjective manner. In this research, the visual distributed debugger prototype is compared against debuggers currently available in an Integrated Development Environment (IDE). A mixture of quantitative estimates (wherever possible) and empirical results is used to support the hypothesis of this research. The research evaluates the system against the program visualization criteria established in Chapter 2.

The design and implementation of the prototype is conducted in a modular manner. Testing of the system takes place in the same way. Initial analysis takes place on the effectiveness of the focus + context techniques. The modified UML displays are compared to those in the original ArgoUML. The overall goal of this part of the research is to maintain the symbology and semantics of UML while improving its space efficiency to accommodate rapid access to both high level and detailed system information. The research analyzes the effectiveness of the techniques with respect to this goal.

The researcher extracts quantitative results based on the number of classes displayable in a set screen area. Empirical discussion is also be generated to support the hypothesis that these visualization techniques better display distributed systems than standard displays.

Evaluation of the effectiveness of the overall debugging system is empirically based. A review of the debugging experience takes place for two different systems. This allows testing of the debugger for two test loads: distributed systems and large systems.

3.4 Experimental Metrics

As with most human-computer interactive activities, it is difficult to quantify the effectiveness of this system under test, in this case, the visual debugger. This section discusses the quantitative and empirical techniques used to gather as much objective evidence as possible and subjective evidence where it is not.

As a result of the type of testing chosen to be conducted, there are only four metrics: the number of viewable classes displayed in a unit area, the empirical evidence gathered in support of the hypothesis as discussed and two types of resource usage data. The empirical evidence includes discussion on the effectiveness of debugging with these techniques including the effects on access time, cognition and the size of portion of the model able to be displayed on a screen. The resource information will include the Central Processing Unit (CPU) utilization as a percent of that available and memory usage by the debugger system.

3.5 System Architecture

This section analyzes the resultant system architecture of the developed prototype. To simplify the development process, modifications are made to existing code where possible. The prototype uses an estimated 500,000 lines of code to reduce the development workload. This re-used code is based on the ArgoUML project. Figure 11 shows a high-level architectural view of the overall system.

This research adds a series of visual modifications, primarily focus + context techniques, to the ArgoUML framework. Chapter 4 discusses these modifications in more detail. The visual modification process retrieves input data from a NovoSoft UML Model, an open-source library for storing UML. Figure 11 also shows the integration of the debuggee JVMs and the connections to them into the rest of the system. JPDA demonstration software forms the basis for the debuggee connection system. Two types of connections to the JVM are available: shared memory and sockets.

The overall system architecture is primarily event based. Observer threads for each virtual machine connection detect modifications to the UML models. These observers then trigger visual modifications and update the screen as required.

As discussed above, the high-level data flow is primarily event based. This section provides a more detailed view of the high-level data flow. Figure 12 shows a high-level data flow model for the entire system. The observer searches for new classes and class information in the debuggee JVM. The UML model is constructed and updated through a reverse-engineering process as information becomes available from the connected JVMs. The UML model is stored in a NovoSoft UML library. The `ExecutionManager` classes control the connections and pass the data to observer threads. These threads add to the

UML model and call visual modification processes when required. The display of the model updates when the threads call the visual modification processes. The user is able to see the resultant view of the software along with source code and debuggee process input and output. The user may also update the display by retrieving further information from the NovoSoft model.

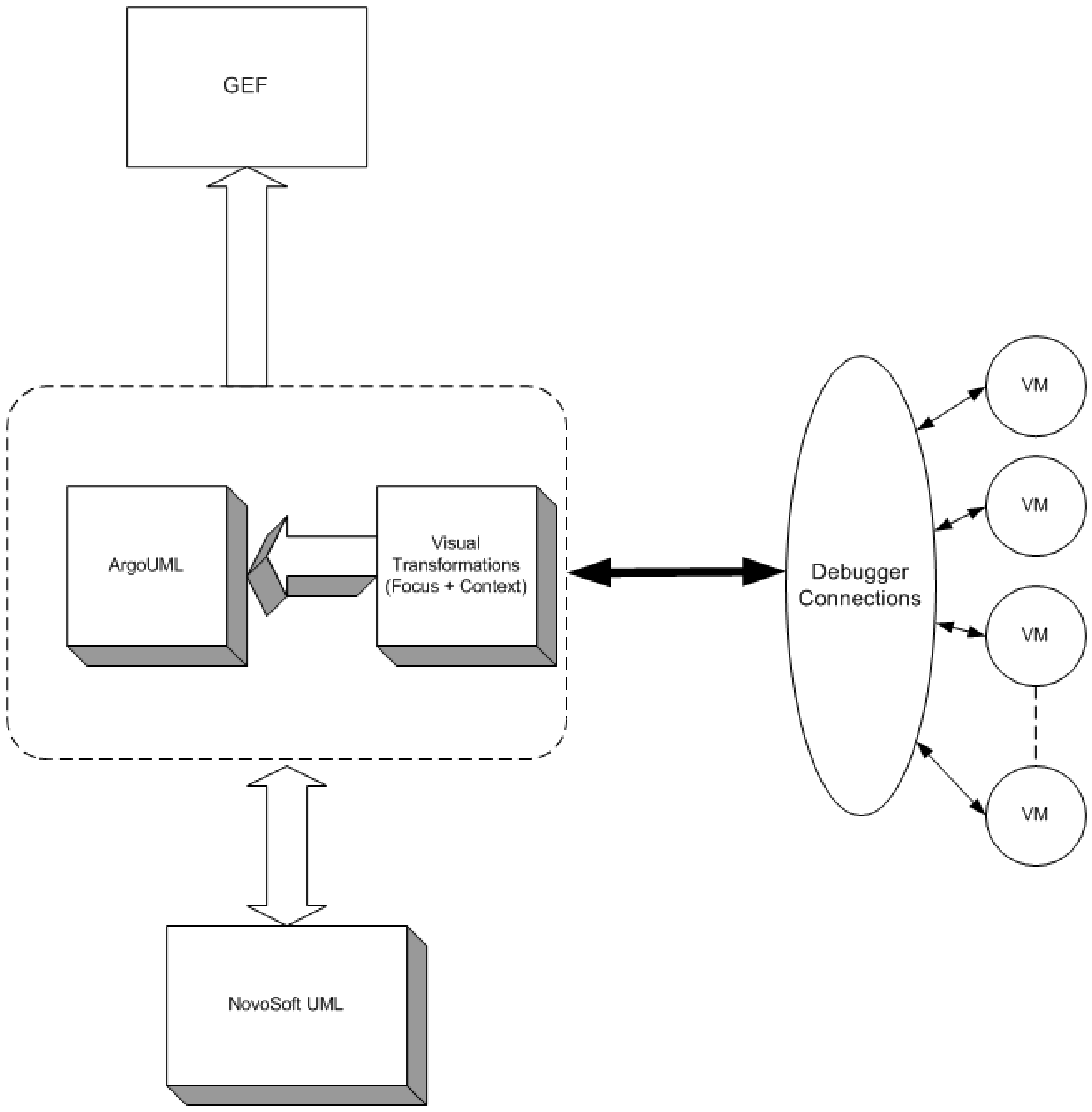


Figure 11. High-level system architecture.

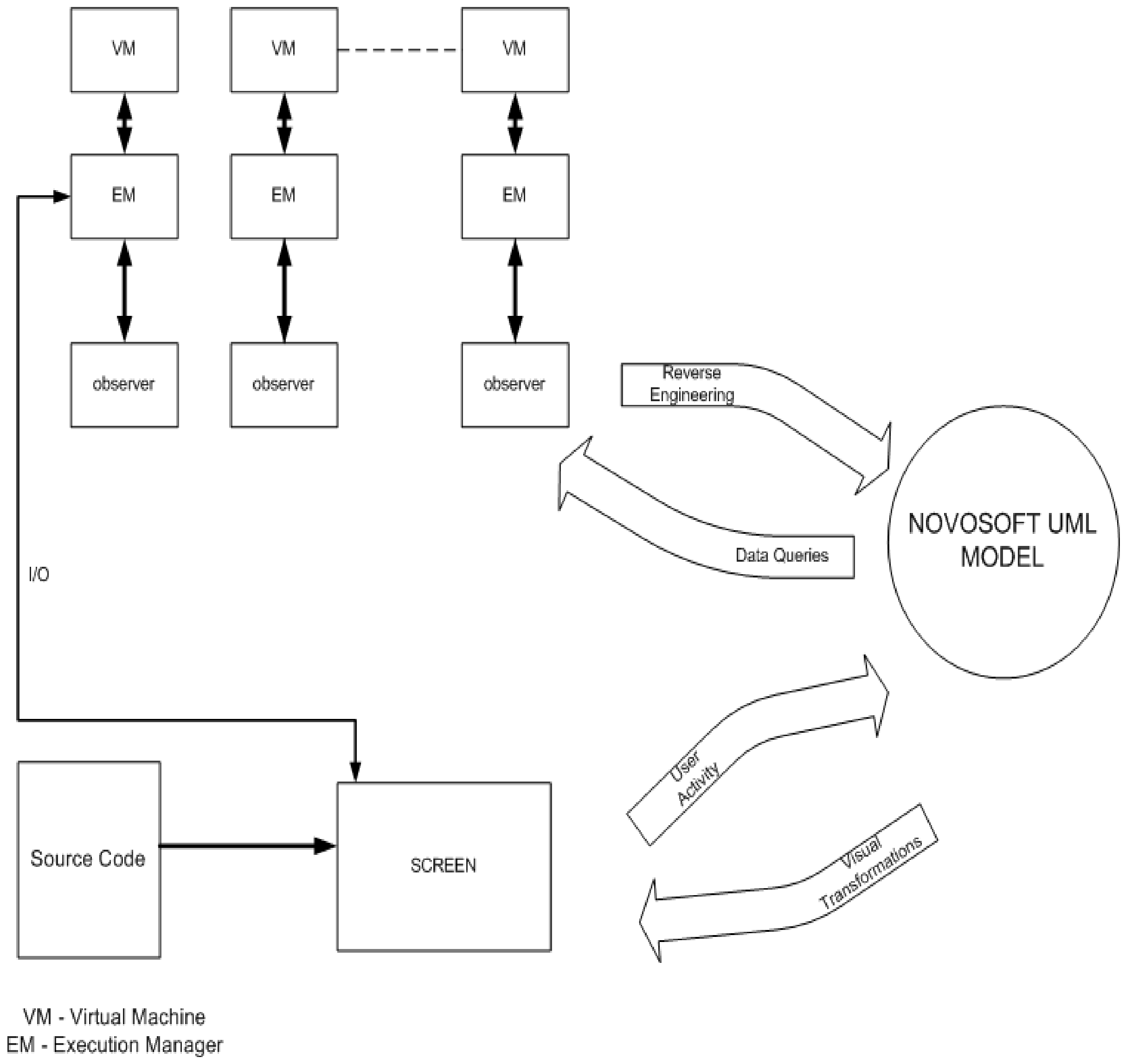


Figure 12. Prototype system data flow model.

Figure 13 presents a pseudo code description of the processing involved for each debuggee process. Chapter 5 gives a more detailed explanation of the executed processes and their order.

```
Connect to JVM
Do
        Do reverse engineering for model
        If model is changed
                Update model
                Update Object diagram on screen
                For each segment
                        Set focal points
                        View transform – Focus + Context
                        Apply Graph layout
While JVM is running
```

Figure 13. Pseudo code for data flow.

3.5.1 ArgoUML

ArgoUML is derived from the Graph Editing Framework (GEF) that provides basic capabilities to display a variety of simple shapes and connect them with links [GEF]. ArgoUML was developed by a separate research project [ARG] and it provides additional functionality to form a CASE tool. During the implementation of the prototype for this research, much of the superfluous ArgoUML functionality has been removed to improve system performance. In particular, separate critiquing and To-Do threads have been removed as they are no longer relevant for this new functionality. The most important parts of ArgoUML for this research are the components responsible for drawing class diagrams. This section explores these components further.

Figure 14 shows the inheritance hierarchy in the figure elements contained in both GEF and ArgoUML. The basic class used to represent a graphical component is Fig. Other classes extend Fig to provide the functionality required for each UML component. In most cases, the functionality can be determined based on the name of the class. Of note is the FigEdgePoly; this class extends a FigEdge which represents a start and end point for a line. The FigEdgePoly class provides a mechanism to draw the line between these points as a series of connected lines to avoid crossings.

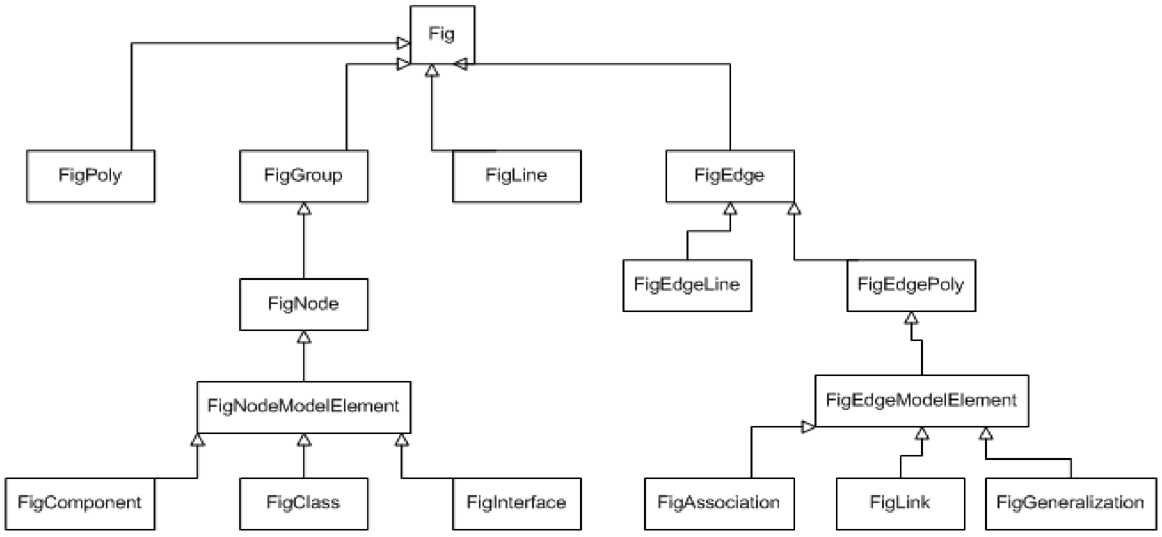


Figure 14. Display figure hierarchy for ArgoUML.

The inheritance hierarchy shown in Figure 15 describes the relationships between classes used to represent UML components. These classes are all contained in the NovoSoft UML library used by ArgoUML.

3.5.2 Debug Interface with JPDA

As discussed, the debugger component in the system is based on JPDA demonstration code. This component establishes a connection to the debuggee JVM which is stored in an execution manager. The JPDA defines an API for accessing data from the JVM. The execution manager maintains methods to extract data from the JVM by using the JPDA API. The user should have a simple means of extracting data from the debuggee program but the execution manager is still too low level for this. To improve data access, the system includes a CommandInterpreter class. The user or a user program can retrieve data from the JVM with simple commands, for example "classes" which returns a list of the classes currently loaded in the JVM. The CommandInterpreter operates as an adapter pattern.

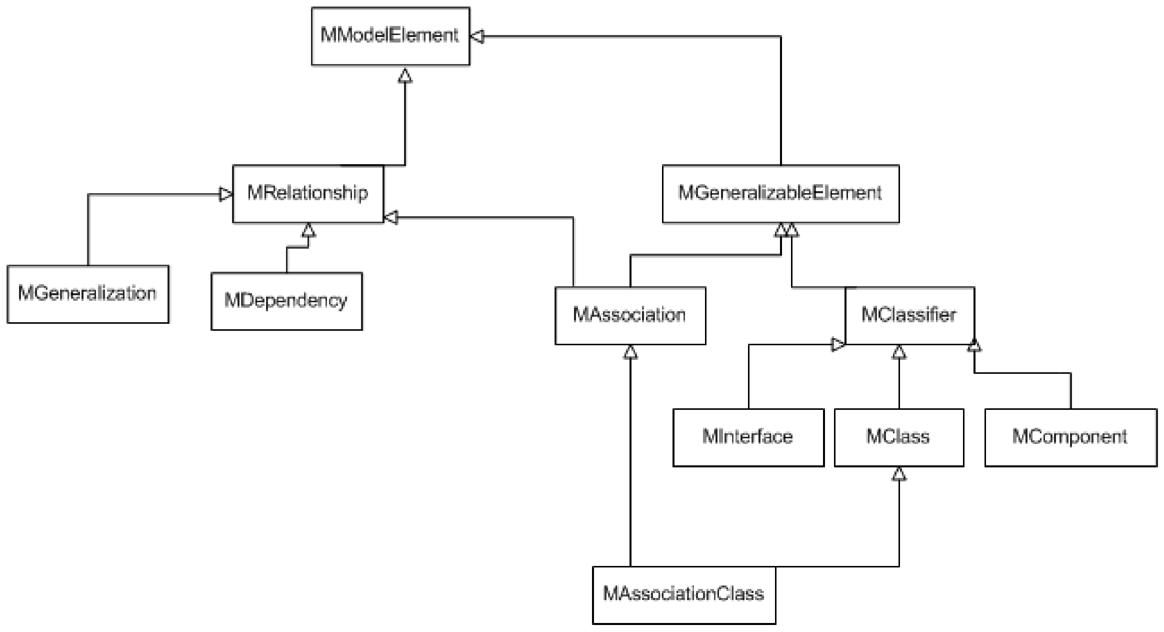


Figure 15. NovoSoft UML hierarchy.

3.6 Summary

This chapter presents the design of the system developed for this thesis research based on the objectives for the visual display and the design objectives for the prototype debugger that is required for testing. This chapter discusses the experimental techniques, which are based on a three part empirical evaluation of the prototype. The debugging experience is discussed for large and distributed systems. Metrics, providing quantitative analysis where possible, are introduced. In addition, this chapter explores the overall architecture of the prototype system including the programs' basis, ArgoUML, and the JPDA.

4 Visual System

4.1 Design

The visualizations derived to facilitate debugging form the basis of this research. Several processes and algorithms are required to meet the visual objectives defined in section 3.2.1. These algorithms allow:

- Specification of a user's Degree of Interest (DOI) in each node in the UML graph, based on user activity within the system and important regions in the graph,
- Information hiding based on the DOI resulting in a variety of Levels of Detail (LODs), and
- Improved space efficiency of the displayed model via the use of graph layout algorithms.

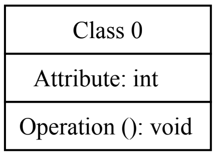
4.1.1 Visual Modifications

Even with the use of a visual language such as UML, the size of many software systems still leads to visual models that are extremely large and complicated. For just a moderately sized system, a typical UML object diagram covers many pages and is too large for efficient navigation of the overall system structure and lower level details. Common solutions to this problem are multi-page printouts, overview windows for navigating the entire diagram, and hypertext links between related sub-models. Multi-page printouts can effectively show overall system structure and individual component details. However, multi-page printouts suffer from many drawbacks such as increased visual scanning, difficult production and management, and lack of support for interaction and dynamic editing. Overview windows and hypertext may provide access to overall system structures and individual component details, but this does not occur in a single view, thus requiring more effort and reducing the effectiveness of the visual representation.

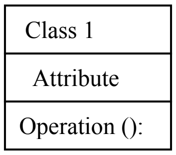
This research proposes another approach to this problem that applies focus + context techniques to an interactive UML diagram-editing tool. Focusing on the UML object diagram, it defines multiple levels of detail for representations of classes and relationships. The system presents an appropriate level of detail representation for each component using a degree of interest function based on frequency of access and distance. Finally, the prototype uses smooth animation to relocate software components within the diagram to emphasize hierarchical relationships while maximizing space economy.

Selective filtering and modified graphical elements are used to create multiple level-of-detail (LOD) representations for UML classes. At the highest level of detail, each class consists of a rectangle divided into three segments that include textual labels for the class name, attributes, and methods. The technique selectively filters the textual labels at lower levels of detail according to the perceived relevance of the information. The levels of detail supported by this solution are summarized below.

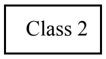
- LOD 0 contains the highest amount of detail. For a UML class representation, this includes a full size graphical representation that includes textual labels for all attributes and methods along with type information.



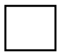
- LOD 1 hides attribute and return types while minimizing margin size.



- LOD 2 only displays the name of the class at a reduced font size.



- LOD 3 removes all textual information and only indicates the presence of a component.



- LOD 4 contains no textual information and indicates the presence of a component at a reduced size.

- LOD 5 completely hides the component from the user.

The technique displays a class at a particular level of detail using a degree of interest (DOI) function based on the frequency of access to a particular class and its distance from the current class in focus (Equation 1).

$$DOI \propto \log_2(freq) + (max\_visible\_doi - dist) \quad (1)$$

where

freq is the number of times the node in question has been accessed by the user;
dist is the graphical distance from the node to the focal point; and
max_visible_doi is the maximum DOI at which a node is visible.

Each time the user accesses a node, the system recalculates the degree of interest and the display updates to reflect the appropriate level of detail for each node. Should a displayed node be further from the focal point than a hidden node, all hidden nodes in between have their LOD increased to 4 to prevent dissection of the graph.

To demonstrate this mapping of LOD representations to the DOI function, consider the sample class diagram in Figure 16. If the user selects “Class 0” as the node of focus, the representations of all remaining classes are modified as shown in Figure 17. “Class 0” is selected as the node of focus so it is displayed with no change at LOD 0. The degree of interest for “Class 1” is one lower than that for “Class 0” so “Class 1” is displayed at LOD 1. As each subsequent class in this diagram is one additional graphical link away from the node of focus, “Class 0”, each subsequent class is represented at the next lower level of detail. Beyond “Class 4” no classes in this path would be shown.

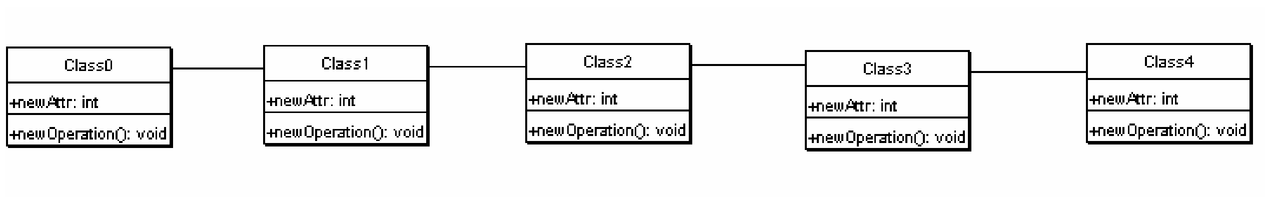


Figure 16. Diagram prior to application of visualization techniques.

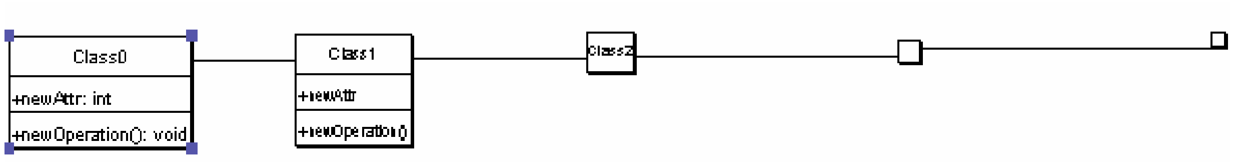


Figure 17. Class diagram following application of focus + context techniques.

Generalization and aggregation are key concepts in object-oriented software systems. These relationships are hierarchical, with a generalization relationship linking a more general class to one or more classes that inherit attributes of the general class. Similarly, aggregation links a container class to one or more classes representing parts that are combined within the container class. As hierarchical relationships, both generalization and aggregation are good candidates for selective aggregation techniques that hide lower levels of the hierarchy. For instance, there may be little value in showing that a car consists of wheels, an engine and chassis, when the user is only interested in seeing that there is a car. UML also includes association relationships that indicate a non-hierarchical relationship between two classes.

Selective aggregation is applied based on UML’s association, aggregation, and inheritance relationships, for those classes that are graphically distant from the class in focus. The display shows the appropriate UML symbology for inheritance or aggregation to indicate that it is hiding classes. For an association relationship, a partial link is displayed indicating hidden classes. When the system brings content into focus, the hidden hierarchy expands revealing the aggregate relationships within.

Figure 18 and Figure 19 show the results of the selective aggregation techniques for a simple class diagram. Figure 18 depicts a class diagram before selective aggregation is applied. The link with a triangle at the end depicts an inheritance relationship and the link with a diamond depicts an aggregation relationship. The display shows the triangle and the diamond positioned at the end of the link associated with the parent class. Figure 19 presents the class diagram after applying the selective aggregation techniques. The display continues to show the triangle, diamond, and a small line to indicate that the hidden classes connect via some relationship to the class. Any classes of a lower DOI and further from the focal point are also hidden from view. The research assumes that if the aggregate class is of little interest and is thereby hidden, then so will those objects that it is associated with.

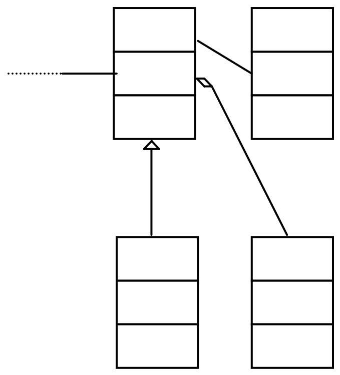


Figure 18. Part of class diagram.

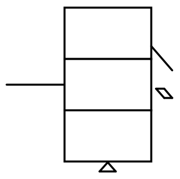


Figure 19. Class diagram after focus + context showing relationships.

4.1.2 Graph Layout

This section first examines the requirements of the graph layout algorithm for this research. Prior research is used as the basis to examine the choice of algorithm and the modifications to it explained. Finally, this section presents an example of applying the algorithm to a graph.

4.1.2.1 Graph Layout Requirements

To maximize the space efficiency of the focus + context techniques for UML, the algorithm modifies the positions of elements in the model. In evaluating how best to accomplish this, a variety of UML graph layout algorithms are examined [AUB, RAT]. A recurring theme in these algorithms is the presentation of hierarchical inheritance relationships down the vertical axis. As this is a commonly used approach and is understood and expected by users, the layout algorithms enforce this presentation approach.

As one applies the layout algorithm to object diagrams following a focus + context transformation, massive differences in node sizes are highly likely. In order to maximize space efficiency, the algorithm considers the size of the nodes. In addition, the algorithm should preserve user context of the system (if any exists) by minimizing the relative reassignment of node positions.

4.1.2.2 Graph Layout Algorithm Development

The main algorithm requirement is to present the graph hierarchically. In particular, the algorithm positions generalization relationships vertically on the display. Layered layout algorithms meet this requirement [BAT99]. These algorithms allow the hierarchical presentation of diagrams with vertices arranged in horizontal layers. The standard method consists of the following steps:

1. Assign a layer to each node
2. Reduce the number of edge crossings
3. Assign horizontal coordinates to each node

Layer assignment requires a process to determine which layer each vertex belongs to. For this research, a layer difference is considered to exist across each generalization relationship, with the top most vertex being the most generalized class. Should a cycle exist in the graph, vertices may be placed on more than one layer. In this case, the algorithm places a node that is a direct descendant of an inheritance relationship at a lower level. All other nodes in the cycle remain at the original level determined through the association relationships. In the simple case where each vertex is the same size, the y-coordinate of each layer is determined by adding a suitable gap to the last layer.

This research does not apply the crossing reduction process because the vertex sizes vary so greatly. With variable vertex sizes, the edge crossing minimization generally achieved from this step will not succeed. The standard algorithm relies on constant row sizes to reduce crossings.

The last step deals with the positioning of each vertex on a layer. The priority for our algorithm is to preserve context rather than minimize edge crossings, so the algorithm allocates the position of each vertex on each row based on its position prior to ordering. The user will see objects in the same relative position as before the modification.

Application of the layout algorithm requires the movement of nodes and links in the object diagram. If done instantaneously, this movement may cause the user to lose their context. To avoid this, the system uses smooth animation when moving graphical elements between their original and revised positions. The technique also moves elements in the graph in two stages to preserve context further. The first stage positions elements in the appropriate level of the inheritance hierarchy. The second stage positions leaf nodes to optimize space efficiency. The system applies the graph layout algorithm after all degree of interest and selective aggregation functions.

4.1.2.3 Graph Layout Example

An explanation and example of the results of applying the visualization features and layout algorithms follows. Figure 20 shows the initial class diagram for this example.

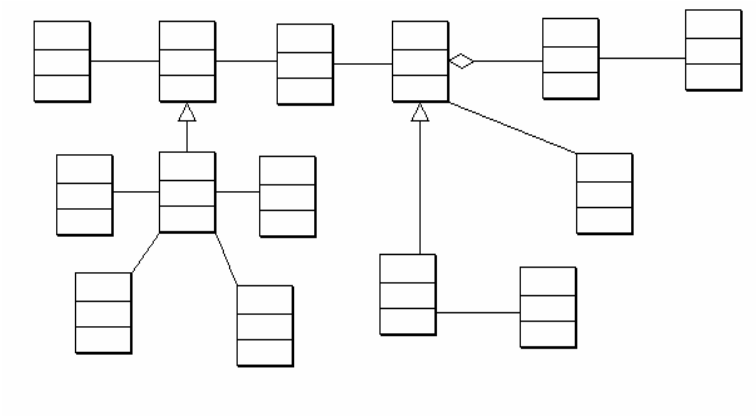


Figure 20. Initial class diagram.

The user enables the focus + context techniques through a context sensitive menu. Figure 21 presents the results of setting the upper left node of Figure 20 as the focal point. Selective aggregation applies to both the aggregation and inheritance relationships from the far right node.

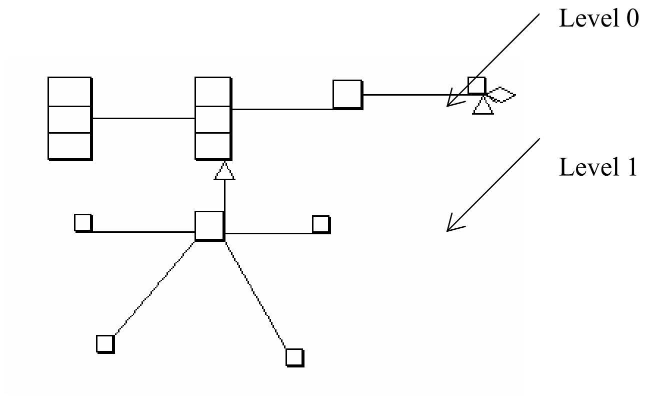


Figure 21. Class diagram with focus + context applied.

Prior to applying the layout algorithm, the hierarchical level of each visible node is determined. A difference of one level exists across an inheritance relationship. Each node connected via an association or aggregation relationship has an identical level. The algorithm places all nodes having the same level on the same row. Figure 21 identifies the level (and row number) of the nodes on each side of the inheritance relationship.

The first stage of the layout algorithm examines each row (starting from the top row) and searches for generalization links from each of the nodes in that row. Figure 21 contains one node with a generalization link. The x position of a node with generalization links is equal to the average of the x positions of the generalized classes in the row above it. In the most common case where a node has a single parent, the algorithm places it directly below. Should this position not be free, the algorithm places it as close as possible to the desired position on the same row.

The algorithm places all remaining nodes on a row without a laid-out position in order based on their x position in the row prior to the layout process. The y position for each row is determined by adding a gap to the lowest point in the above row. This stage of the algorithm has the effect of shrinking the overall display area yet it maintains the general layout of the original diagram. Figure 22 shows the results of this process.

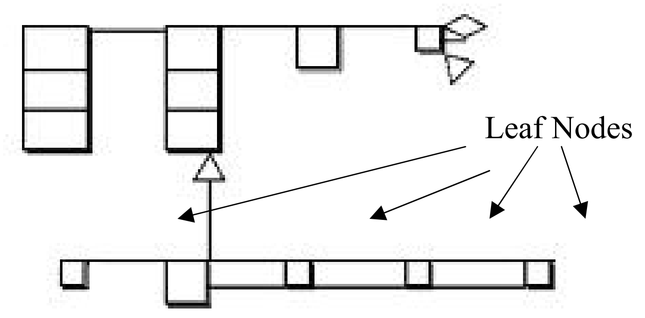


Figure 22. Class diagram after phase 1 of layout algorithm.

The second stage of the layout algorithm searches each row for nodes with leaf nodes on the same row. If these leaf nodes have a relatively low LOD (and vertical height), they are shifted to the right side of the associated node. The algorithm arranges leaf nodes radially with the radius proportional to the number of leaf nodes. If possible, the position of other nodes on that row is left unchanged. Should there be insufficient room, then all nodes to the right of the leaf nodes are shifted right to establish a suitable gap.

The bottom left node of Figure 22 has four associated leaf nodes on its row. The second stage of the algorithm arranges these leaf nodes in a radial fashion by as shown by Figure 23. This part of the algorithm allows a greater number of nodes to be displayed than otherwise possible while still preserving user context. This algorithm is particularly effective when placing leaf nodes with a low LOD around a node with a high LOD.

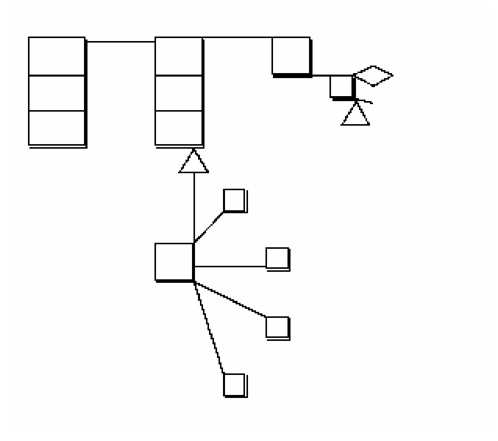


Figure 23. Class diagram after phase 2 of layout algorithm.

The process repeats for each segment of the graph when there are multiple unconnected segments in the model. The algorithm places each segment at the top of the screen and to the right side of the previous graph segment.

4.2 Implementation

This section further explains the design choices involved in integrating the software visualization features into the prototype system. To determine the most effective design for integrating the visualizations into ArgoUML, the processes that must occur are considered. This research develops a fisheye lens that displays less detail for components with a smaller degree of interest and applies selective aggregation techniques to hide components that are beyond a specified degree of interest. Finally, a graph layout algorithm arranges components to emphasize hierarchical relationships while maximizing space efficiency.

The modifications introduced are largely to do with presentation of the graph nodes and links, i.e. should they be painted and if so, is their presentation varied in some manner? As such, large parts of the changes occur in the objects that represent the graphical elements for classes and the various UML link types. The classes representing the display of these objects include FigClass, FigEdgeModelElement, FigAssociation, FigGeneralization. No changes are necessary to the NovoSoft UML library that stores the model information.

The ClassDiagramGraphModel class maintains a list of the figure elements to display for an object diagram, making it a suitable location for the modification methods. The ClassDiagramGraphModel class controls most of the fish-eye view change. It calculates the hierarchical level of each FigClass (the class that represents the image of a class), the DOI and LOD for each, and finally calls a "change" method to effect the visual modifications. The ClassDiagramGraphModel class also determines how each link should be drawn, that is either fully displayed, only the arrow, a partial line, or not at all. If a FigClass has hierarchical or aggregate relationships, the class also determines whether the branches for these relationships should collapse.

Debugger events and mouse events in the FigClass class trigger the alteration process. These two events largely control the visualization process. As such, this module of the overall system is event driven. The "change" method within this class paints the object based on its LOD value.

4.3 Summary

The hypothesis for this research states that visualizations can be of assistance to users in distributed and large system debugging. Based on this hypothesis, the visualizations are of great importance. This chapter defines the algorithms and processes developed to produce these visualizations. It employs selective filtering to produce altered class representations based on user activity and a nodes' distance from the focal point. The visual system applied selective aggregation to inheritance and aggregation relationships to reduce clutter by objects considered less important to the user. The research develops a layered graph layout algorithm to preserve user context and UML semantics while increasing space efficiency. This chapter also discusses the implementation of the visual system in ArgoUML.

5 Debugger System

5.1 Design

The debugger system comprises several main features to extract and transform the data, and provide the standard debugging features required by users. The following sections discuss these topics with additional information provided on implementation.

5.1.1 Data Extraction

Chapter 2 discusses the need to minimize the effects caused from observing a system. Failure to do this may cause the monitor to obtain inaccurate results. JPDA allows system data extraction directly from the observed JVM requiring no modifications to the original code. Thus, some reduction in processing speed is expected to occur, however synchronization points and all data values remain unchanged from an unmonitored system. The system gives the user the option to connect via one of two methods as shown in Figure 24: launch a new JVM with specified arguments or attach to an existing JVM and communicate via a socket. A GUI demonstration application for JPDA is available with the latest versions of the JDK. This application has been modified to interface with the modified ArgoUML system for this research.

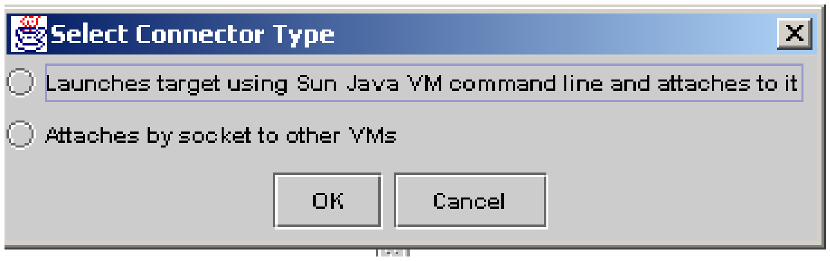

Select Connector Type
Launches target using Sun Java VM command line and attaches to it
Attaches by socket to other VMs
OK
Cancel

Figure 24. Connector launch options.

5.1.2 Reverse Engineering

Once connected via the JPDA, raw data is available from each monitored JVM. The basic premise behind this research, however, states that the user can debug more effectively with the data presented in a visual format. The most effective representation for this data is a UML object diagram, as this is a commonly used and accepted standard. To produce this UML model, the system applies a reverse-engineering process. The following discusses some of the issues involved in this process.

Once the user creates a connection within ArgoUML to the debuggee process, the JVM sends a signal notifying the debugger application that it is now running. The program counter at this point is at the start of the program and it has not executed any commands. As such, when the debugger accesses the JVM data, nothing is present. The only effective way to obtain data appears to be a delay of several seconds before data extraction. To circumvent this problem, a separate observer thread is created within ArgoUML. The observer queries the JVM at set intervals while execution of the debuggee process is not suspended. This solution also improves system performance. If the debuggee process executes unhindered, then the JVMs will provide a large amount of data to the debugger and the display will not be able to update in time.

Once the observer detects new classes in the JVM, the reverse-engineering process examines them for operations and fields where a field contains the name and type of an attribute. The observer constructs a NovoSoft UML model from each of the added classes. The observer also adds association, aggregation and inheritance relationships to the model as it detects these relationships. Figure 25 describes the algorithm for the reverse engineering process.

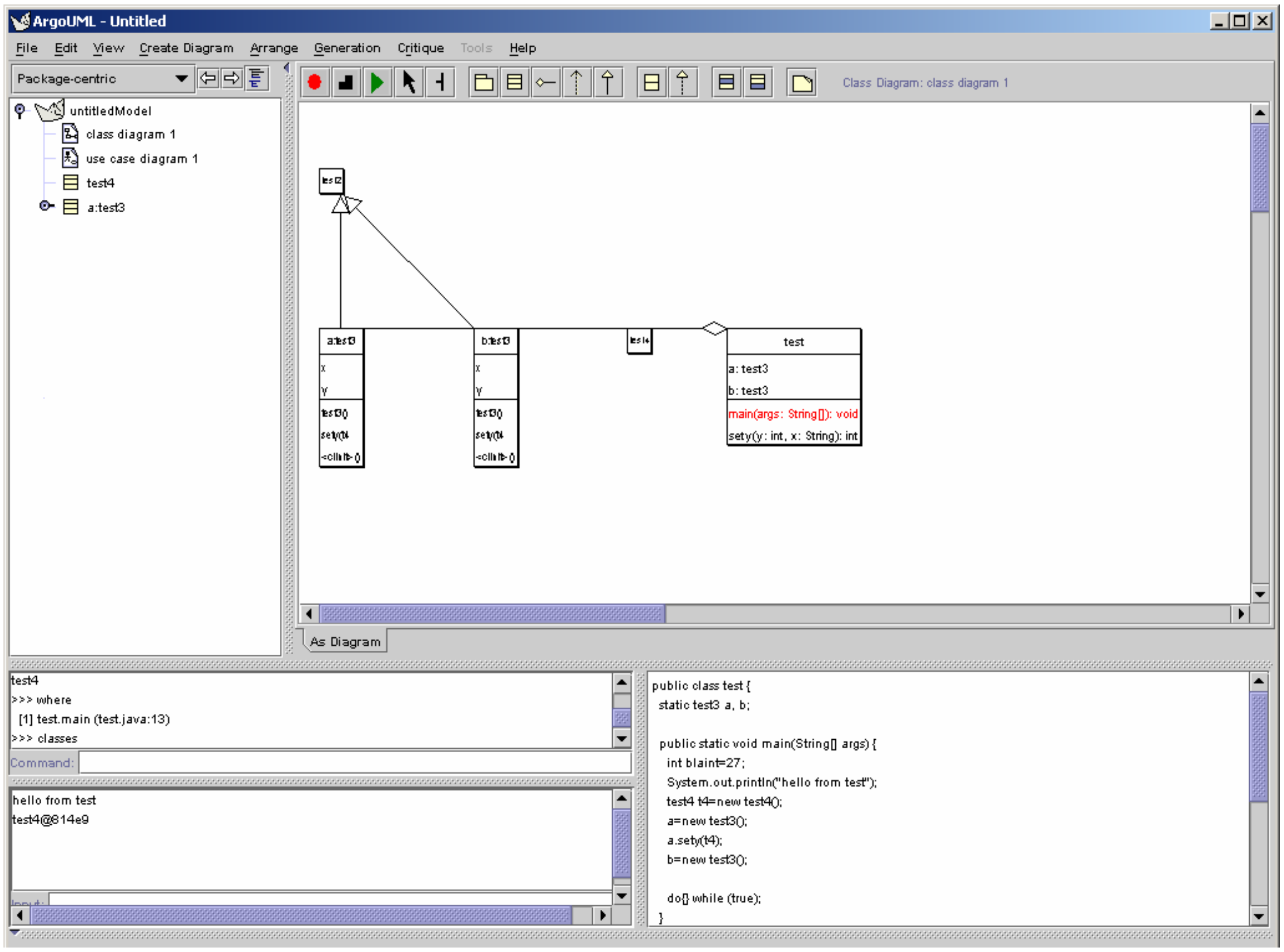
```
For each class found
    If class not in model
        Get fields
        Get Methods
        For each field
            Add attribute
            Create an object of type (attribute)
            Build aggregation to the object
        For each method
            Add method information to the class
            If the method parameter type is contained in the model
                Build an association from the class to the parameter type
For each class in the model
    If there is a generalization relationship to an existing type in the model
        Add the generalization
```

Figure 25. Pseudo code for Reverse Engineering algorithm.

5.1.3 Debugger

Figure 26 shows a screenshot of the program while connected to a simple test program. The top-right window presents the object diagram view in standard UML format. The top-left view presents a hierarchical list of each node and link in the diagram. The user can select elements in this list for rapid access to nodes with a low DOI.

The debugger sub-system is largely based upon example code provided by Sun with version 1.3 (and later) of the JDK. Research into the operation of this code found that it is also largely event driven. The requirements of this component are that it communicates with the GUI to request or display information provided by the debuggee JVM. This usage scenario suggests that an event driven interface might also be effective between the GUI and the debuggee process.


ArgoUML - Untitled
File Edit View Create Diagram Arrange Generation Critique Tools Help
Package-centric
Class Diagram: class diagram 1
untitledModel
class diagram 1
use case diagram 1
test4
a:test3
test
a: test3
b: test3
main(args: String[]): void
sety(y: int, x: String): int
As Diagram
Command:

```
test4
>>> where
  [1] test.main (test.java:13)
>>> classes
```

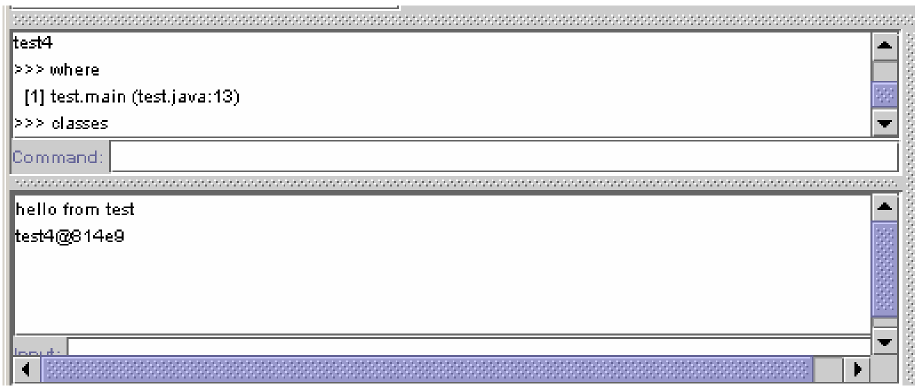
```
hello from test
test4@814e9
```

```
public class test {
  static test3 a, b;

  public static void main(String[] args) {
    int blaint=27;
    System.out.println("hello from test");
    test4 t4=new test4();
    a=new test3();
    a.sety(t4);
    b=new test3();

    do{} while (true);
  }
```

Figure 26. Debugging screen layout.

An observer thread handles communication between the debuggee process and the GUI. This thread periodically accesses the JVM to determine if there any changes to the observed system. If there are changes, an event triggers the reverse engineering process and updates the model on the screen.

A variety of commands may be issued via a text input facility as shown by the "Command" prompt in Figure 27. A list of listeners captures output from the debuggee processes and displays this output as shown in the bottom window of Figure 27. The user can enter command line input to the debuggee processes in the bottom input section of the lower window in Figure 27.



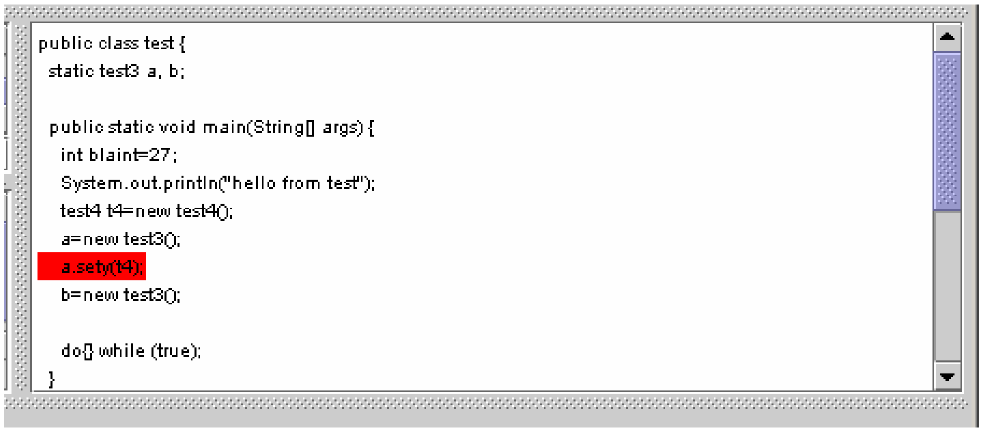
```
test4
>>> where
  [1] test.main (test.java:13)
>>> classes
```

```
hello from test
test4@814e9
```

Figure 27. Debuggee process I/O window.

To be effective as a visual debugging system, access to the source code should be provided (although it is not required to provide a run-time model of the debuggee process). The ability to set and cancel breakpoints via direct manipulation of the source code is also highly desirable. The system allows breakpoints to be set and removed by a double-click of the mouse on the appropriate line. The observer queries the execution manager to determine if the breakpoint is valid. Figure 28 shows highlighting of valid breakpoints in red. The research chose red due to its cultural association with stopping (for example traffic lights). The source code for a class with a “main” method is automatically loaded when the process begins.

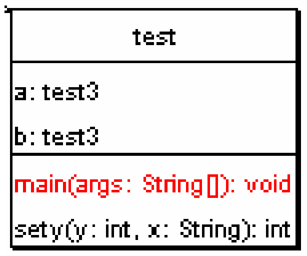
When the debugger is operating, the focus of each segment of the graph (corresponding to a software model reverse-engineered from a JVM) is initially set to the class containing the “main” method. Each segment of the graph corresponds to a single debuggee process. Research considers this appropriate because it is the commencement point for execution in each system and is often integral to the architecture of a system.



```
public class test {
  static test3 a, b;

  public static void main(String[] args) {
    int blaint=27;
    System.out.println("hello from test");
    test4 t4=new test4();
    a=new test3();
    a.sety(t4);
    b=new test3();

    do{} while (true);
  }
```

Figure 28. Breakpoint setting in source code.

To monitor and debug a system, it is also highly desirable to highlight the current execution point, as this requires less cognitive effort than manual tracking. This allows the user to follow execution at a higher level of abstraction rather than follow the execution through each line of code. The ability to follow program execution provides the user with an understanding of the order of executed components and some insight into system behavior. The debugger highlights the current method of execution for each debuggee process in the object diagram as shown in Figure 29.



```
test
a: test3
b: test3
main(args: String[]): void
sety(y: int, x: String): int
```

Figure 29. Execution point display.

This visual debugging system allows the user to control the execution of the debuggee process as do most other debuggers. That is, controls exist to provide for unrestricted execution (the green play button), stepping through the code line-by-line (step), or temporary cessation of execution (a red circle) as shown in Figure 30. If connections to multiple JVMs exist, the user must select a component from the segment of interest prior to selecting the control. The listener then determines which component to select and carries out the appropriate action in the selected JVM.

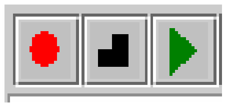


Figure 30. Debugger controls.

5.2 Implementation

The design and architecture presented described how the debugger works to a great extend. The following provides additional guidance on the implementation of this component.

5.2.1.1 Patterns

To allow a flexible and easily maintainable interface between the debugger component and ArgoUML, patterns are used. An adapter pattern in the **CommandInterpreter** class allows the observer to issue simple commands without an underlying knowledge of the operation of the system. The **ExecutionManager** class maintains knowledge of the JVM parameters.

5.2.1.2 Connections

The debugger can only establish shared memory connections with JVMs on the same physical machine. A debugger can connect with a socket to a JVM almost anywhere. There is extra overhead associated with a socket connection. As such, users should use a shared memory connection when they launch an application on a local machine. Due to security restrictions with Java, it is not possible to launch an application on a separate machine (via a socket).

5.2.1.3 Data Extraction and Reverse Engineering

The raw data from the JVM includes JDK base classes and the full names of all data types, for example “java.lang.String”. The full name of such classes and the display of all loaded classes in the reverse-engineered class diagram add unnecessary clutter to the diagram. The reverse engineering process truncates known data types to remove this clutter, for example, “String”. It also filters JDK base classes from the class diagram.

5.3 Summary

In order to evaluate the effectiveness of the techniques described in Chapter 4, the researcher modifies ArgoUML to include these transformations and additional debugging capabilities. The research interfaces ArgoUML to the debuggee JVM via JPDA and other debug code. The class diagram of the model for the display is reverse-engineered from data accessed in the debuggee JVM. This section also discusses debugger functionality including breakpoint setting, debugger controls and implementation details.

6 Results And Analysis

6.1 Introduction

Previous chapters have discussed prior work in this field and the formation of the prototype system. This chapter presents the results of the experimental techniques discussed in Chapter 3 and analyzes the system's effectiveness based on these results.

6.2 Collection of Results

As the debugger system testing relies on analysis of other programs, the selection of these programs is highly critical in obtaining meaningful results. Ideally, the research would test the modified ArgoUML with several large modules connected and operating as a single distributed system. Due to the difficulties in setting up such a test, this research took a different approach. The research conducts testing in three phases to evaluate the system against each type of objective: the generic visual display ability, and debugging of a moderately large system and a distributed system with several components. The following sections discuss the testing performed for each of these areas and present the results. No other applications run concurrently with the debugger or debuggee programs to affect system performance.

6.3 Experimental Setup

Three experiments are conducted, each with the same hardware and operating system configuration. Each PC in use for the testing contains a Pentium 4 2GHz processor and 1 GB RAM running the Windows 2000 operating system. The AFIT computer network connects these PCs with 100Mb/sec Ethernet Network Interface Cards.

In all cases, the debugger and each debuggee process run on their own PCs. The tests use Java Runtime Environment v1.3.1 on each PC.

Several command line parameters are required when initiating a process to provide access to debugger programs through the JPDA. The user can initiate debuggee programs in either a suspended or an unsuspended state. Suspension of the program causes execution in the JVM to stop prior to executing any instructions. In an unsuspended state, the program will execute as normal, however the user can monitor or debug the program when it stops at the point where some user input is required. Debugging in an unsuspended state does not work well with GUI input. The GUI threads are unable to execute as normal while in this state and cannot receive mouse input. All tests use sockets to establish a connection. Figure 31 shows a typical set of command line parameters to enable debugging.

```
        java -Xdebug -Xnoagent -Xrunjdwp: transport = dt_socket, server=y,
suspend=y, address=14422 test
```

Figure 31. Command line options for execution.

These command-line options allow for JPDA socket access on port 14422. The "test" class executes in the JVM.

6.3.1 General Display Testing

If one assumes simple classes such as those in Figure 16, then roughly 24 classes fit on a single screen. By applying the degree of interest and graph layout algorithms, one can fit approximately 75 classes on a single screen. Thus, the system provides three times the space efficiency without even resorting to selective aggregation or radial arrangement of leaf nodes. The degree to which selective aggregation increases space efficiency is dependent on the level of fan out in the system relationships.

Access to high level and detailed system information is more difficult to evaluate. With original representations, the user will need to scroll through multiple pages if there are more than 24 classes. In order to deal with large systems, the user would have to commit much of the overall system information to long-term memory. By applying the focus + context techniques, 75 classes can be readily viewed simultaneously with the structure of hundreds of classes potentially available using selective aggregation. This enables the user to offload cognitive processing to the external visual presentation.

Using original ArgoUML representations, details are readily available for up to 24 classes at a time. However, the user can still only focus on one area of the screen at a time. With the focus + context view, only a few components are in focus at any given time. The user needs to select another component to bring it into focus. Once they bring the appropriate component into focus, the time to access detailed information is similar to that of the original UML. If the system displays all components that the user is likely to access on the same page, it is likely to be faster to switch focus using the original UML. However, for the common case where components spread over multiple pages in the original UML, the focus + context techniques are likely to improve access time since more components are accessible from the same screen.

6.3.2 Large System Testing

For this part of the testing, the debugger system analyzes Bubble World [VAN02], an AFIT research system. Bubble World is a visual information retrieval system designed to test novel information retrieval techniques. Bubble World consists of 84 classes and approximately 34,000 lines of code.

Reverse engineering provides the user with an object diagram instantaneously eliminating the need to form a mental model by browsing through much of the code to identify the classes, methods, attributes and relationships. By debugging with the modified UML, one can readily observe the flow of control between instances of classes along with the static structures indicating how the various classes are related. If one can fit 50 lines of code on a page, 34,000 lines of code would require 680 pages. Based on the theoretical calculations for UML described in section 6.3.1, 84 classes would require between three and four pages to display with typical UML presentation styles. With the modified UML presentation, nearly 90 percent of the classes should fit on a single page. In actuality, due to many attributes and methods in the classes in this system, less than two classes fit on a page with standard UML presentation techniques as shown in Figure 32. The modified UML presentation displays nine classes on a single page as shown in Figure 33. This number exceeds theoretical expectations slightly, however, there is room for improvement. There is a significant amount of white space in the diagram reducing the space efficiency of the system.

The graph layout algorithm described is most effective at improving space efficiency where the node sizes are similar. This, however, is rarely the case for class diagrams. Further arrangement of leaf nodes for the largest node reduces the total surface area used by the diagram. The revised algorithm places these leaf nodes vertically down the screen while their total height is less than that for the very large node. Figure 34 shows the results of the modification to the algorithm. All nine classes are still visible, yet the modification reduces the total area required for the display. Altering the arrangement in this way reduces user context, as these leaf nodes are no longer in the same relative position.

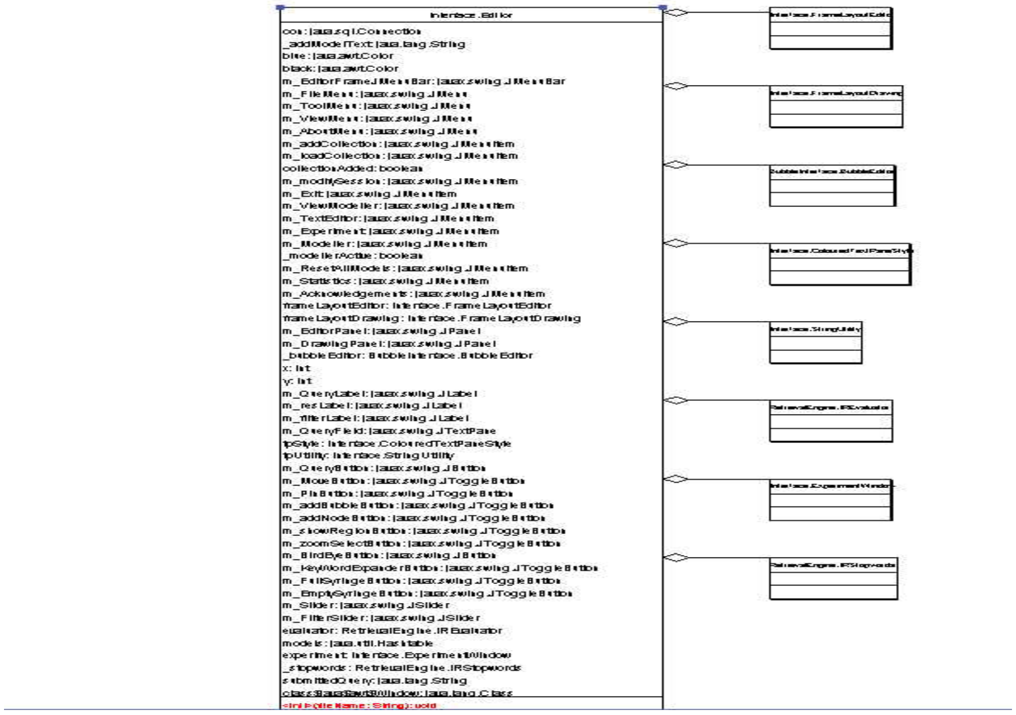


Figure 32. Bubble World model without focus + context.

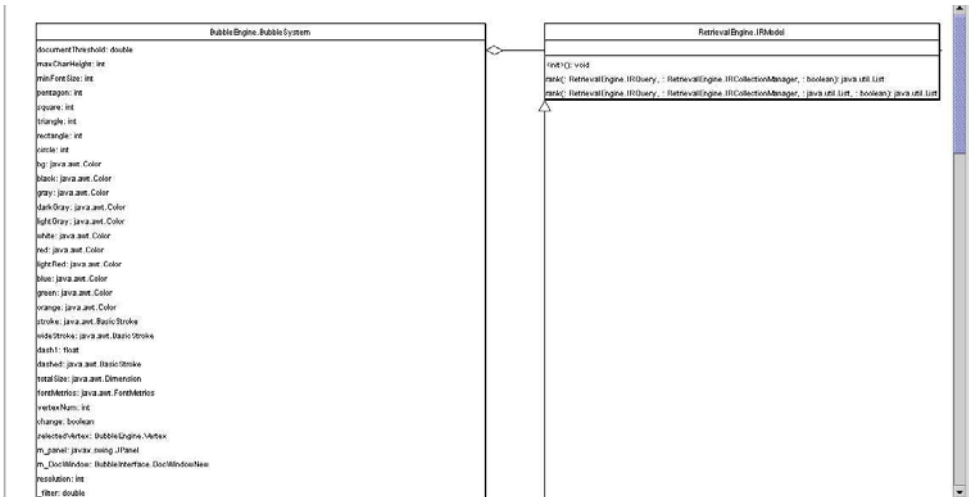

BubbleEngine.BubbleSystem
documentThreshold: double
maxCharHeight: int
minFontSize: int
pentagon: int
square: int
triangle: int
rectangle: int
circle: int
bg: java.awt.Color
black: java.awt.Color
gray: java.awt.Color
darkGray: java.awt.Color
lightGray: java.awt.Color
white: java.awt.Color
red: java.awt.Color
lightRed: java.awt.Color
blue: java.awt.Color
green: java.awt.Color
orange: java.awt.Color
stroke: java.awt.BasicStroke
wideStroke: java.awt.BasicStroke
dash1: float
dashed: java.awt.BasicStroke
totalSize: java.awt.Dimension
fontMetrics: java.awt.FontMetrics
vertexNum: int
change: boolean
selectedVertex: BubbleEngine.Vertex
m_panel: javax.swing.JPanel
m_DocWindow: BubbleInterface.DocWindowNew
resolution: int
_filter: double
RetrievalEngine.IRModel
<init>(): void
rank(: RetrievalEngine.IRQuery, : RetrievalEngine.IRCollectionManager, : boolean): java.util.List
rank(: RetrievalEngine.IRQuery, : RetrievalEngine.IRCollectionManager, : java.util.List, : boolean): java.util.List

Figure 33. Bubble World with focus + context applied.

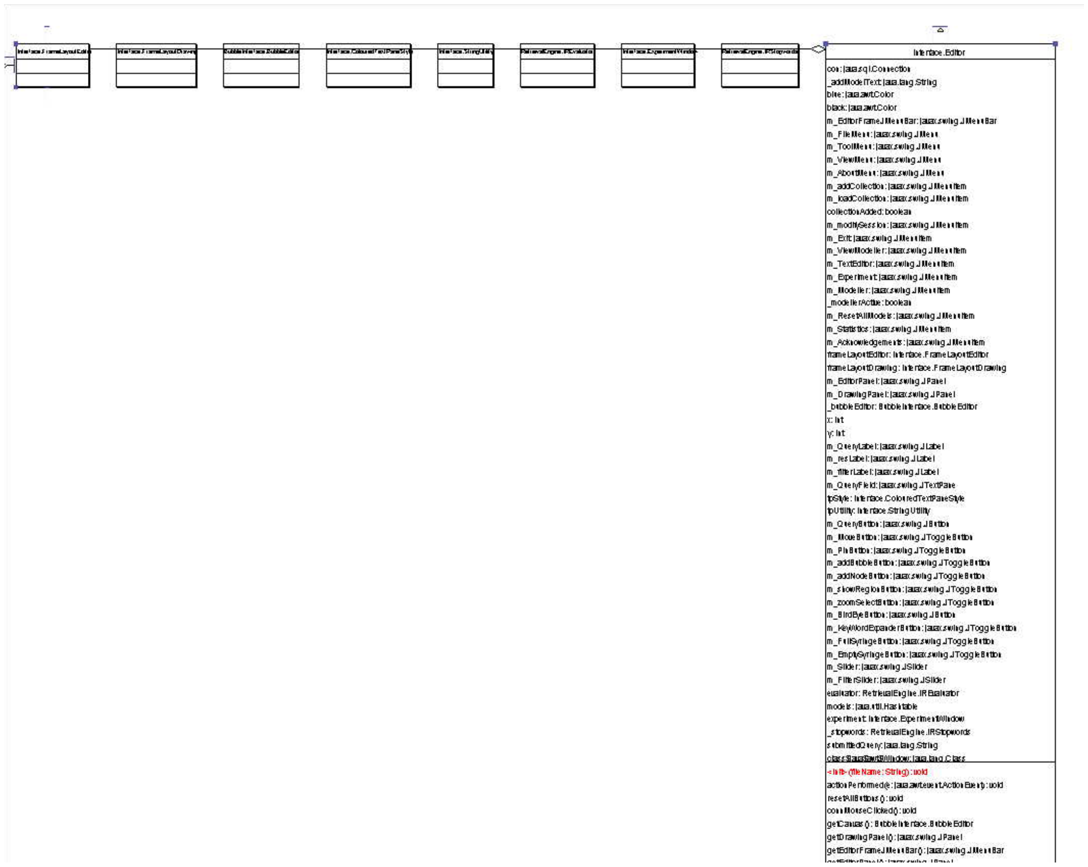

Interface.FrameLayoutEditor
Interface.FrameLayoutDrawing
BubbleInterface.BubbleEditor
Interface.ColouredTextPaneStyle
Interface.StringUtility
RetrievalEngine.IREvaluator
Interface.ExperimentWindow
RetrievalEngine.IRStopwords
Interface.Editor
con: java.sql.Connection
_addModelText: java.lang.String
blue: java.awt.Color
black: java.awt.Color
m_EditorFrameJMenuBar: javax.swing.JMenuBar
m_FileMenu: javax.swing.JMenu
m_ToolMenu: javax.swing.JMenu
m_ViewMenu: javax.swing.JMenu
m_AboutMenu: javax.swing.JMenu
m_addCollection: javax.swing.JMenuItem
m_loadCollection: javax.swing.JMenuItem
collectionAdded: boolean
m_modifySession: javax.swing.JMenuItem
m_Exit: javax.swing.JMenuItem
m_ViewModeller: javax.swing.JMenuItem
m_TextEditor: javax.swing.JMenuItem
m_Experiment: javax.swing.JMenuItem
m_Modeller: javax.swing.JMenuItem
_modellerActive: boolean
m_ResetAllModels: javax.swing.JMenuItem
m_Statistics: javax.swing.JMenuItem
m_Acknowledgements: javax.swing.JMenuItem
frameLayoutEditor: Interface.FrameLayoutEditor
frameLayoutDrawing: Interface.FrameLayoutDrawing
m_EditorPanel: javax.swing.JPanel
m_DrawingPanel: javax.swing.JPanel
_bubbleEditor: BubbleInterface.BubbleEditor
x: int
y: int
m_QueryLabel: javax.swing.JLabel
m_resLabel: javax.swing.JLabel
m_filterLabel: javax.swing.JLabel
m_QueryField: javax.swing.JTextPane
tpStyle: Interface.ColouredTextPaneStyle
tpUtility: Interface.StringUtility
m_QueryButton: javax.swing.JButton
m_MoveButton: javax.swing.JToggleButton
m_PinButton: javax.swing.JToggleButton
m_addBubbleButton: javax.swing.JToggleButton
m_addNodeButton: javax.swing.JToggleButton
m_showRegionButton: javax.swing.JToggleButton
m_zoomSelectButton: javax.swing.JToggleButton
m_BirdEyeButton: javax.swing.JButton
m_KeyWordExpanderButton: javax.swing.JToggleButton
m_FullSyringeButton: javax.swing.JToggleButton
m_EmptySyringeButton: javax.swing.JToggleButton
m_Slider: javax.swing.JSlider
m_FilterSlider: javax.swing.JSlider
evaluator: RetrievalEngine.IREvaluator
models: java.util.Hashtable
experiment: Interface.ExperimentWindow
_stopwords: RetrievalEngine.IRStopwords
submittedQuery: java.lang.String
class$javaStartWindow: java.lang.Class
<init>(fileName: String): void
actionPerformed(e: java.awt.event.ActionEvent): void
resetAllButtons(): void
conMouseClicked(): void
getCanvas(): BubbleInterface.BubbleEditor
getDrawingPanel(): javax.swing.JPanel
getEditorFrameJMenuBar(): javax.swing.JMenuBar

Figure 34. Bubble World with modified layout algorithm.

The debugger establishes a connection to Bubble World via socket. A processing cycle consists of data extraction, reverse engineering, model creation, display and application of the graph layout. The first processing cycle takes 15.4 seconds. Each later cycle takes 0.15 seconds. During the first processing cycle, CPU utilization remains close to 100 percent for the entire 15 seconds. After the initial processing, CPU utilization reaches approximately 20 percent each time the debuggee process is re-examined (approximately 4.5 seconds). The first processing cycle takes longer and has greater CPU utilization as it extracts a considerable amount of data when compared to later processing cycles. The initial graph layout will also take considerably longer than others will, as objects are not displayed near their final positions, requiring extensive animation. Bubble World consumes 16,000KB of memory while the debugger application requires 64,912KB.

The debug controls allow the user to step through the program while tracing the execution on the UML class diagram. The increase in the LOD for objects as they execute is valuable for allowing tracking through this relatively large diagram.

As discussed, only nine instances of classes are initially visible. Most of these instances do not have their method and attribute information available. As the program executes further and loads these classes into memory, the reverse-engineering process updates the model and display with the information for these classes. The user may watch the new instances spawn. The layout algorithm is re-applied when new classes are added to the diagram. In general, the addition of new classes results in a smoothly animated sequence. Minor adjustments to the layout are all that is required to insert the new class. Figure 35 shows a view of the debugger later in the execution of the program. Eight classes are now visible in the diagram.

The space efficiency of the diagram is re-examined after applying the modified layout algorithm. Thirteen instances are now visible as shown in Figure 36 providing a 150% improvement in space efficiency. Although no formal evidence is available, usage suggests that the loss of context for the modification is minimal.

Debugging with breakpoints is effective as the program stops directly at the user's point of interest. Debugging a suspended program by stepping through each line of code without the aid of breakpoints is a slow process due to the large number of instructions to be stepped through. One can increase the speed of stepping by stepping up through the execution tree (by entering a command in the command window). Stepping up completes execution of the current method and resumes debugging one process higher in the stack.

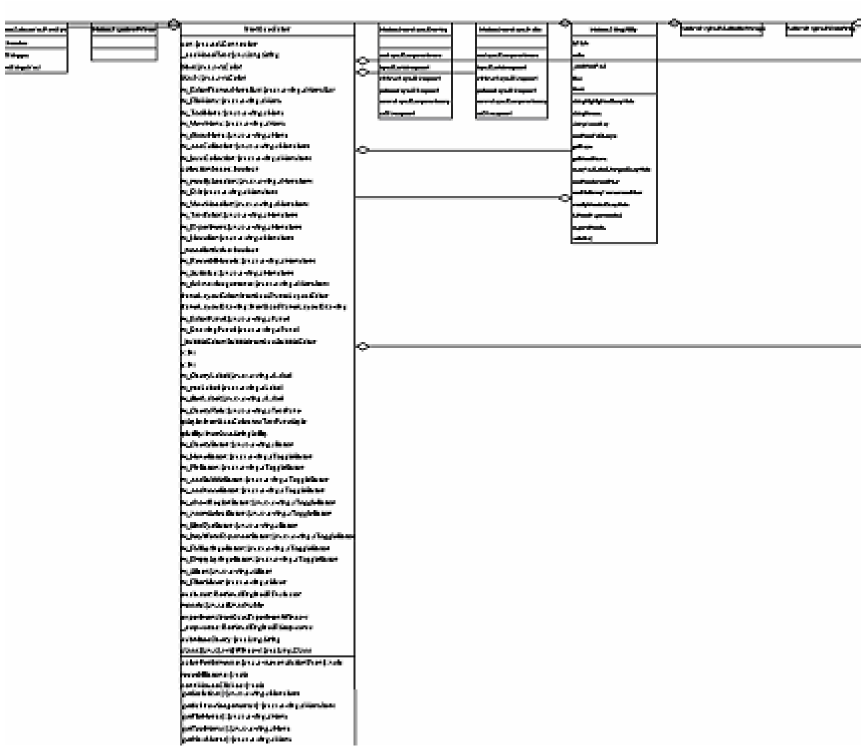


Figure 35. Later in execution showing large differences in vertex sizes.

6.3.3 Distributed System Testing

Kil developed visual techniques for analyzing the execution and performance of distributed agent systems [KIL02]. In order to evaluate a prototype system implementing these techniques, a small agent-based command and control system was developed. Each agent is constructed of four classes and approximately 1,000 lines of code. The test spawns five different types of agents and attaches them to the debugger.

The debugger system is attached in turn to each agent process. First processing times ranged from 0.6 seconds for the first attached process to 4 seconds for the fifth process. CPU utilization remained at approximately 100 percent for all first processing cycles. However, once the system had stabilized, CPU utilization remained below approximately 25 percent. Memory requirements increase from 15.6MB when monitoring one agent to 57Mb when the debugger is monitoring all five agents.

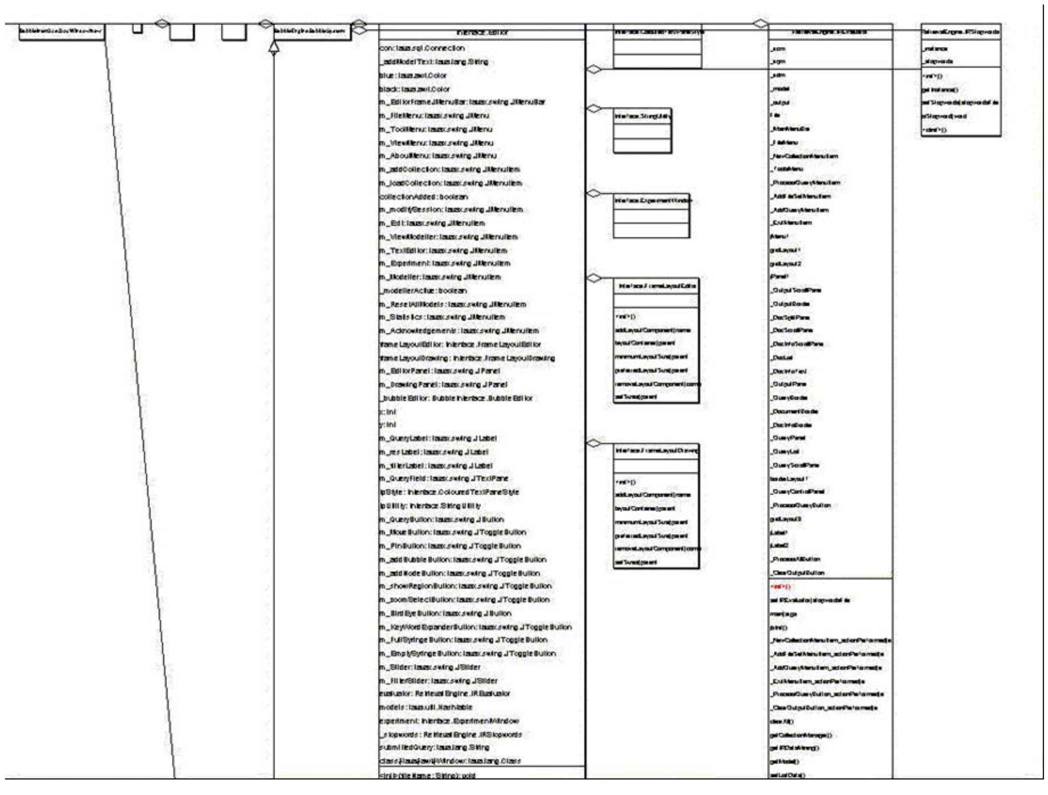


Figure 36. New graph layout later in execution.

Without the visualization techniques developed in this research, the debugger can only display parts of two agents on a single screen as shown in Figure 37. With these visualization techniques, the display is able to fit information about each of the five processes in a single screen as shown in Figure 38. The number of displayed instances increases from seven to 20. The ability to see all processes in one screen is considered a big advantage over traditional debugging systems. If an agent is not of interest, the user cannot control the display to reduce the space consumed by that system. It would be beneficial to have the option to display a system name rather than its contents. For example, assume the command and control system is operating with five processes and four are of the same type "Infantry" but the suspected error is in the "Logistics" system. The user may prefer to reduce the number of instances displayed from 20 to eight. They still have detailed access to the "Logistics" agent and can see the processes interacting with it at a higher level of abstraction.

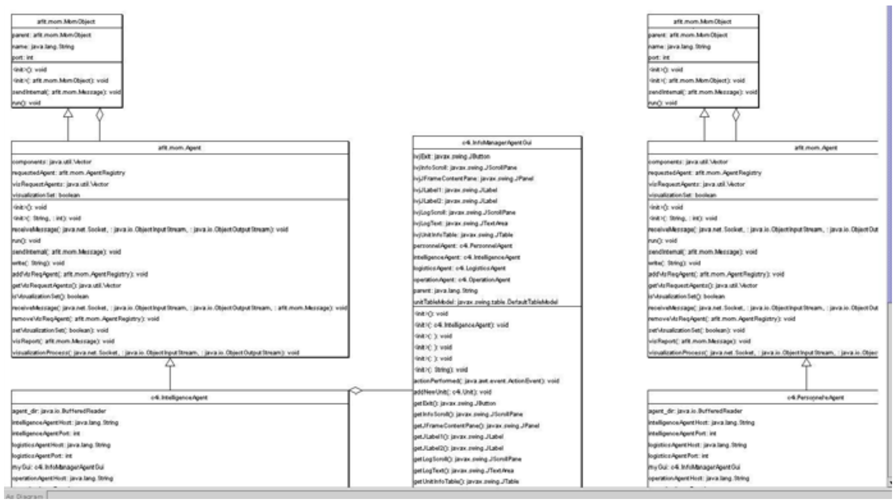


Figure 37. Distributed system without visualization.

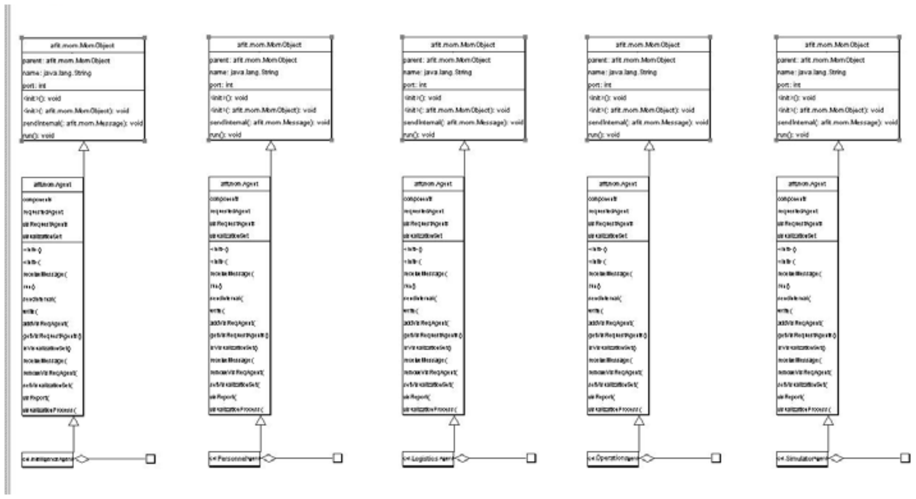


Figure 38. Distributed system with visualization.

If inter-process communication triggers a method invocation in another system, as is often the case, then the display indirectly shows this communication. Monitoring of inter-process communication is difficult to control due to the program execution speeds available, that is either full speed execution or manual stepping. If the debugger could step automatically, at user-defined speeds then inter-process communication would be far easier to monitor.

6.4 Analysis of Results

It is a trivial process to show the improvement in space efficiency provided by the techniques in this research. The theoretical results demonstrate at least a 300% (75/24) improvement in the number of instances, the large system achieved a 650% (13/2) improvement and the distributed system achieved a 286% (20/7) improvement. This clearly demonstrates that the focus + context techniques provide significant improvement in space efficiency over standard UML layouts for a class or instance diagram. As expected, the results from actual systems at least match the theoretical results due to the large size of "real" classes. The distributed system did not exceed the theoretical prediction due to the low number of classes and the small length of chains within the graph. As such, there was only a small reduction in the average size of each displayed node.

A revised layout algorithm is introduced which offers space efficiency improvements of 163% (13/8) for the large system. There is a reduction in preservation of user context as the leaf nodes placed vertically next to the large nodes are no longer in the same relative position. However, usage suggests the reduction in context is less significant than the improvement in space efficiency

One can analyze the results further using the program visualization criteria described in section 2.4.1.2 – scope, abstraction, specification method, interface, presentation, fidelity and invasiveness. The current implementation limits the scope of the visualization techniques to that provided with the UML class diagram along with source code. The system does not yet allow presentation of watch values on the class diagram display. The use can identify program structure while monitoring the flow of control as the program executes. Similarly, abstraction is primarily achieved through the visual representations supported by UML. One can view the application at different levels of detail that include overall system structure, classes and relationships, methods and attributes, and source code. The JVM provides specification methods that require little or no input from the user. In a distributed environment, the user will have to specify the Internet address of the application and source code. No changes are required to any of the application code, thus the only way these techniques might alter the behavior of the application is through the consumption of resources on the computer, CPU and memory, required for the second JVM. The user can reduce the effects of this problem by running the debugger program on a separate platform to the debuggee application.

The UML implementation in ArgoUML defines the user interface and presentation semantics. The system modifies the presentation to make it more efficient using focus + context while the semantics are preserved.

The resource requirements of the debugger system are initially quite high. However, once initial processing is complete the requirements reduce significantly allowing for consecutive debugging of large and distributed programs on networked PCs. The system appears to be able to debug larger systems based on the resource requirements for the tested systems.

6.5 Summary

This chapter reviews the experimental procedures used for obtaining results to support the research hypothesis. It discusses the environment and basic setup for three experiment types to analyze the three main components of the prototype: visual modifications, large system analysis and distributed system analysis.

The results are analyzed using resource requirements and visual effectiveness. The system provides an improvement of 650% for large systems and 286% for distributed systems exceeding the theoretical prediction of a 300% improvement. A modified layout algorithm is devised to achieve improved space efficiency and a small reduction in user context. In addition, the system is analyzed based on program visualization evaluation techniques developed by Roman and Cox and Price et al. This evaluation suggests the techniques used are promising with one of the main benefits being the ability to view the system at multiple levels of detail.

The system aids debugging for the user by increasing space efficiency. This increase provides a greater understanding of the relationship between "interesting" classes in the diagram and the remainder of the system. This improvement also applies to distributed systems. The user is able to see interactions between the various components in the system with reduced scrolling.

7 Conclusions and Recommendations

7.1 Introduction

This chapter reviews the problem definition for this research. It revisits the requirements along with the methodology used to test the developed hypothesis. Finally, the researcher suggests future developments for this topic.

7.2 Research Review

This section reviews the need for this research, methodology and its success at meeting the research goals.

7.2.1 Background

The JBI is a large distributed environment linking many types of applications and databases to users over a variety of protocols. Debugging the JBI or any distributed system requires both knowledge of the overall system structure and detailed knowledge of relevant aspects of the debuggee system to determine the exact cause of any problems. For large systems, this becomes difficult. For distributed systems such as the JBI, the complexity of the problem increases even further.

A variety of debugging systems are available to deal with distributed systems, however, few are capable of dealing with all of the required operating systems with a low-intrusion data extraction process. Existing systems often force users to debug a program by examining lines of code.

This research hypothesizes that appropriate visualization methodologies reduce the complexity involved in the debugging process by offloading the user's mental model of the program structure and interaction to the display. Specifically, this research provides

UML instances diagrams presenting the detected system. This research considers UML instances diagrams to be appropriate for this purpose as software engineers commonly use similar class diagrams for system design. Thus, they do not require additional knowledge and the framework is considered suitable by those in the field. The class diagram is reverse-engineered from data extracted with JPDA from each JVM. Extraction of the model using the JPDA ensures that there is no effect on the debuggee program at compile-time and minimal effect at run-time. The user can reduce the effect on debuggee system performance by running the debugger on a different machine than the debuggee programs.

Even with moderately sized systems, the UML instances diagram may take up many pages. This research introduces a focus + context system that aims to provide detail for those objects which are considered interesting to the user while maintaining access to the overall context in which the detail exists. This allows the overall system structure to be displayed in a much smaller area while still preserving UML notation. The visualization system uses selective filtering and selective aggregation for inheritance and aggregation relationships. The system's space efficiency improves with a modified layered graph layout algorithm.

7.2.2 Research Impact

The effectiveness of the developed methodology is tested by implementing the features in a modified version of ArgoUML. The presentation of UML class and instance diagrams with the focus + context system provides a vast improvement in the number of classes that can be displayed in specified area (up to 650%) while preserving the requirements of UML notation. The improvement in space efficiency allows the user to debug large systems more easily based on their improved knowledge of system structure while maintaining their access to detail information.

The research applies the debugger to a moderately sized research system. Standard debug functionality such as breakpoint setting and various forms of stepping are available. The debugger prototype enables display of system structure along with control flow. The focus + context system displays much more of the system representation on the screen than is otherwise possible.

The debugger is also applied to a small-scale distributed research system. The user can monitor all parts of the system and control each independently with the debugger controls. The display simultaneously shows the control flow of each process providing a basic framework for distributed debugging.

Although this research did not use the prototype to detect errors in either system, evidence suggests that the devised methodology is an improvement over traditional debugging systems. Further tools are required to enhance the prototype for synchronized debugging of distributed systems.

The visual and debugger objectives systems are considered throughout the design and implementation processes. All defined objectives have been met.

7.3 Future Developments

Throughout the development process, a number of refinements to the methodology have been identified which would further enhance distributed debugging. This section discusses some of these refinements.

Evidence suggests the focus + context system is effective at providing detail when it is most likely to be required. The transformations in this process can be improved, in particular:

- Varied LODs based on user testing of the relative importance of each type of information displayed in UML class diagrams;
- Display of watch data with the model;
- More effective representations for large numbers of instances of a class;
- Decaying effect of user activity with time; and
- Selective aggregation applied to packages using clustered graphing techniques [EAD00]. This would allow the system to hide the contents of entire systems if the details are unimportant.

Use of the system suggests that additional debugging controls would be of benefit. These include intelligent stepping as is available with Borland JBuilder and a more easily accessible watch specification and monitoring system [BOR]. The intelligent stepping system jumps over methods in Sun and Java libraries to the code the user is most likely trying to reach. In addition, the ability to control the speed of execution would be advantageous. The user could specify the execution rate with controls similar to a video recorder allowing them to quickly step through less interesting stages of execution and slow the execution for those that are important. This would be particularly useful if a suitable location for a breakpoint cannot be determined due to inadequate knowledge of system execution flow.

With distributed debugging, it is important to have knowledge of network performance. This would allow the user to determine quickly if communication links or machines are not operating correctly. Ideally, the display would combine the class diagram with a deployment diagram.

Finally, support for multiple threads within a virtual machine is highly desirable. This capability is not currently present but is essential as many programs, in particular GUI based systems, operate with several threads of control. The ability for the user to monitor all of them is required for effective debugging of any such program.

7.4 Summary

Distributed and large software systems, including the JBI, are difficult to debug. This research attempts to make this process easier and less time consuming for the user by introducing a variety of visualization techniques into the debugging process. The debugger monitors the debuggee program through the JPDA framework and reverse engineering provides the user with a UML class diagram. The user is able to use standard debugging controls with the debuggee program. The debugger highlights the current method and class of execution on the class diagram allowing the user to track control flow. With large systems, the UML class diagram may itself be too large to use efficiently.

Focus + context techniques enable views of large quantities of information, in this case UML class diagrams. This research applies a fisheye lens with filtering and selective aggregation, to class diagrams. The debugger maintains the symbology and semantics of UML while increasing space efficiency. These visualization techniques improve access to the information, thereby enabling the user to take in more information in the same amount of time. To improve user understanding of the underlying software system, access is provided to both high-level structural information and detail. Furthermore, a graph layout algorithm is provided that organizes software components to emphasize hierarchical relationships while improving space efficiency.

These visualization techniques combine to provide a more effective means of visually debugging large and distributed systems. The user maintains access to detailed information and overview of the system while tracking execution. This aids the user in understanding program operation and allows more effective debugging with the standard debugger controls.

Bibliography

[AMA99] Amari, H. and M. Okada, "A Three-Dimensional Visualization Tool for Software Fault Analysis of a Distributed System", *Proceedings of the IEEE Conference, Man, Systems and Cybernetics*, 4:194-199 (October 1999).

[APP93] Appelbe, W., J. Stasko, and E. Kraemer, *Applying Program Visualization Techniques to Aid Parallel and Distributed Program Development*, Technical report GIT-GVU-91-08, Georgia Institute of Technology, 1993.

[ARG] ArgoUML User Manual, http://argouml.tigris.org/documentation/defaulthtml/manual/, 20 June 2002.

[AUB] Auburn University, Graphical Representations of Algorithms, Structures, and Processes (JGRASP), http://www.eng.auburn.edu/grasp/ 18 September 2002.

[BAT99] Battista, G., P. Eades, R. Tamassia, and I. Tollis, *Graph Drawing- Algorithms for the Visualization of Graphs*. Upper Saddle River: Prentice Hall, 1999.

[BER77] Bertin, J. *Graphics and Graphic Information-Processing*, translated by Berg, W. and Paul, S. New York: Walter de Gruyter Inc, 1981.

[BOR] Borland JBuilder v 7.0 Personal edition. Borland Software Corporation, 18 September 2002.

[BOW94] Bowman, D., A. Ferrari, B. Schmidt, M. Schmidt, B. Topol, and V. Sunderam, *The Conch network concurrent programming system*, Technical report, Emory University, Atlanta, GA, January 1994.

[CAR99] Card, S., Mackinlay, J. and B. Schneiderman, *Readings in Information Visualization – Using Vision to Think*, San Francisco: Morgan Kauffman Publishers, 1999.

[COO99] Cook, S., and S. Brodsky, *OMG Analysis & Design PTF UML 2.0 REQUEST FOR INFORMATION Response from IBM*, Online Document, http://cgi.omg.org/docs/ad/99-12-08.pdf 1999.

[EAD00] Eades, P. and M.L. Huang. "Navigating Clustered Graphs Using Force Directed Methods," *Journal of Graph Algorithms and Applications*, 4:157-181 (no 3, 2000)

[FUR81] Furnas, G.W. "The Fisheye View: A New Look at Structured Files", in *Readings in Information Visualization – Using Vision to Think*, S. Card et al, editors, San Francisco, Morgan Kauffman Publishers, 1981.

[GEF] GEF Documentation, http://gef.tigris.org/project_docs.html, 20 June 2002.

[HAN02] Hangal, S. and M. Lam, “Tracking Down Software Bugs Using Automatic Anomaly Detection”, *Proceedings of the 24th International Conference on Software Engineering ACM*, pp291 – 301, May 2002.

[HEN96] Hennessy, J. and D. Patterson, *Computer Architecture – A Quantitative Approach*, 2nd ed., San Francisco CA: Morgan Kauffmann Publishers, 1996.

[JAI91] Jain, R., *The Art of Computer Systems Performance Analysis*, New York: John Wiley & Sons Inc., 1991.

[JER97] Jerding, D., J. Stasko, and T. Ball, “Visualizing Interactions in Program Executions”, *International Conference on Software Engineering*, 1997

[JON02] Jones, J., M. Harrold, and J. Stasko, “Visualization of Test Information to Assist Fault Localization” *Proceedings of the 24th International Conference on Software Engineering,* 467 –477, ACM Press, May 2002.

[JPD] Java Platform Debugger Architecture Overview, Sun http://java.sun.com/products/jpda/doc, 28 April 2002.

[KAN95] Kan, S., *Metrics and Models in Software Quality Engineering*, Reading Ma: Addison-Wesley Publishing, 1995.

[KIL02] Kil, C.K., *Visual Execution Analysis for Multiagent Systems.* MS Thesis, AFIT/GCS/ENG/02-12. School of, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, Month 2002 (AD-).

[KOT01] Köth, O and M. Minas, “Abstraction in graph-transformation based diagram editors” *Second International Workshop on Graph Transformation and Visual Modeling Techniques.* Vol 50, Elsevier Science Publishers, 2001.

[KRA97] Kraemer, E., “Visualizing Concurrent Programs”, in *Software Visualization: Programming as a Multimedia Experience*, John Stasko et al, editors, The MIT Press, Cambridge MA, 1997.

[LEU94] Leung, Y.K., M.D. Apperley, “A review and taxonomy of distortion-oriented presentation techniques”, *ACM Transactions on Computer-Human Interaction*, 1(2):126-160, 1994.

[MEH00] Mehner, K. and A. Wagner, “Visualizing the Synchronization of Java-Threads with UML”, In *Proceedings of the IEEE International Symposium on Visual Languages*, 2000.

[MIL56] Miller, G., “The Magical Number Seven, plus or minus two: Some limits on our capability for processing information”, *Psychological Science,* 63, 1956.

[OMG00] Object Management Group (OMG), Inc, Unified Modeling Language (UML) Specification Version 1.4, http://www.omg.org/technology/documents/formal/uml.htm

[PAI00] Paige, R., J. Ostroff, and P. Brooke, "Principles for Modeling Language Design", *Information and Software Technology*, 42: 665-675, July 2000, UK.

[PRI97] Price, B., R. Baecker, , and I. Small, "An Introduction to Software Visualization", in *Software Visualization: Programming as a Multimedia Experience*, John Stasko et al, editors, The MIT Press, Cambridge MA, 1997.

[RAT] RATIONAL ROSE, Rational Software Corporation, http://www.rational.com/products/rose/index.jsp 18 September 2002.

[ROM] Roman, G.C., and K.C. Cox, "A Taxonomy of Program Visualization Systems", *IEEE Computer*, December, 1993.

[SHN98] Shneiderman, Ben. *Designing the User Interface. Strategies for Effective Human-Computer Interaction.* Third Edition, Reading MA: Addison-Wesley, 1998.

[SIM90] Simmons, M. and R. Koskela, *Performance Instrumentation and Visualization*, ACM Press - Frontier Series, Addison-Wesley Publishing, p46, 1990.

[STA90] Stasko, J., Tango "A Framework and System for Algorithm Animation", *IEEE Computer*, Vol. 23, No. 9 1990.

[STA98] Stasko, J. et al, *Software Visualization-Programming as a Multimedia Experience*, Cambridge MA: The MIT Press, 1998.

[SUN] The Java Platform Debug Architecture FAQ, Sun, www.java.sun.com/products/jpda/faq.html, 14 October 2002.

[TAN02] Tanenbaum, A. and M. Van Steen, *Distributed Systems Principles and Paradigms*, Upper Saddle River NJ: Prentice Hall, 2002.

[TEL01] Telles, M. and Y. Hsieh, *The Science of Debugging*, Scottsdale AZ: The Coriolis Group, 2001.

[TOP94] Topol, B., J. Stasko, and V. Sunderam, *Integrating Visualization Support Into Distributed Computing Systems*, Georgia Institute of Technology, GIT-GVU-94-38, October 1994.

[USA99] United States Air Force Scientific Advisory Board. *Report on Building the Joint Battlespace Infosphere*, Volume 1, December 17 2000.

[USA] "Usability Evaluation", www.pages.drexel.edu/~zwz22/UsabilityHome.html, 04 December 2002

[VAN02] Van Berendonck, C.L., *Bubble World - A Novel Visual Information Retrieval Technique*. MS Thesis, AFIT/GCS/ENG/02M-09. School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2002 (AD-).

[WAR00] Ware, C. *Information Visualization – Perception for Design*, San Diego CA: Morgan Kauffman Publishers, 2000.

[WIC92] Wickens, C.D. *Engineering Psychology and Human Performance*, 2nd ed., New York: Harper Collins, 1992.

[WON01] Wong, A., T. Dillon, I. May and W. Lin. "A Generic Visualization Framework to Help Debug Mobile-Object-Based Distributed Programs Running on Large Networks," Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (IEEE), 240 (January 2001).

Vita

Flight Lieutenant Benjamin R. Musial joined the Royal Australian Air Force in June 1996 as an undergraduate engineering student. He attended the University of Queensland at St. Lucia in Brisbane and graduated with Honors in a Bachelor of Engineering degree in Computer Systems in 1998. After completing studies with the University of Queensland, he was posted to RAAF Edinburgh, SA. In January 1998, he received his commission and attended the Royal Australian Air Force Officer Training School (OTS) at Point Cook near Melbourne.

Following graduation from OTS, Flight Lieutenant Musial attended several basic Avionics Engineering courses. Following these courses, he was employed as a design engineer in the Maritime Patrol Logistics Management Squadron at RAAF Base Edinburgh near Adelaide.

In August 2001, Flight Lieutenant Musial entered the Air Force Institute of Technology as a graduate student in the computer science and engineering department. He graduated the institute with a Master of Science degree in Computer Engineering.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)* 10-03-2003	**2. REPORT TYPE** **Master's Thesis**	**3. DATES COVERED** *(From – To)* Mar 2002 – Mar 2003

4. TITLE AND SUBTITLE UML Assisted Visual Debugging for Distributed Systems	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Musial, Benjamin R., Flight Lieutenant, RAAF	**5d. PROJECT NUMBER** If funded, enter ENR #2001001
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 Hobson Street, Building 640, Wright Patterson AFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCS/ENG/03M-12
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR / Software and Systems, Attn: Robert L. Herklotz, Ph.D., 801 N. Randolph St., Rm 732, Comm: (703) 696-6565, Arlington VA 22203-1977 e-mail: Robert.herklotz@afosr.af.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFOSR/Software and Systems
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The DOD is developing a Joint Battlespace Infosphere, linking a large number of data sources and user applications. To assist in this process, debugging and analysis tools are required. Software debugging is an extremely difficult cognitive process requiring comprehension of the overall application behavior, along with detailed understanding of specific application components. This is further complicated with distributed systems by the addition of other programs, their large size and synchronization issues. Typical debuggers provide inadequate support for this process, focusing primarily on the details accessible through source code. To overcome this deficiency, this research links the dynamic program execution state to a Unified Modeling Language (UML) class diagram that is reverse-engineered from data accessed within the Java Platform Debug Architecture. This research uses focus + context, graph layout, and color encoding techniques to enhance the standard UML diagram. These techniques organize and present objects and events in a manner that facilitates analysis of system behavior. High-level abstractions commonly used in system design support debugging while maintaining access to low-level details with an interactive display. The user is also able to monitor the control flow through highlighting of the relevant object and method in the display.

15. SUBJECT TERMS

Software Engineering, Debugging (Computers), Distributed Data Processing, Visual Aids,

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Lt Col Timothy M. Jacobs
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	U	112	(937) 255-6565 x4279; e-mail: Timothy.Jacobs@afit.edu

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18